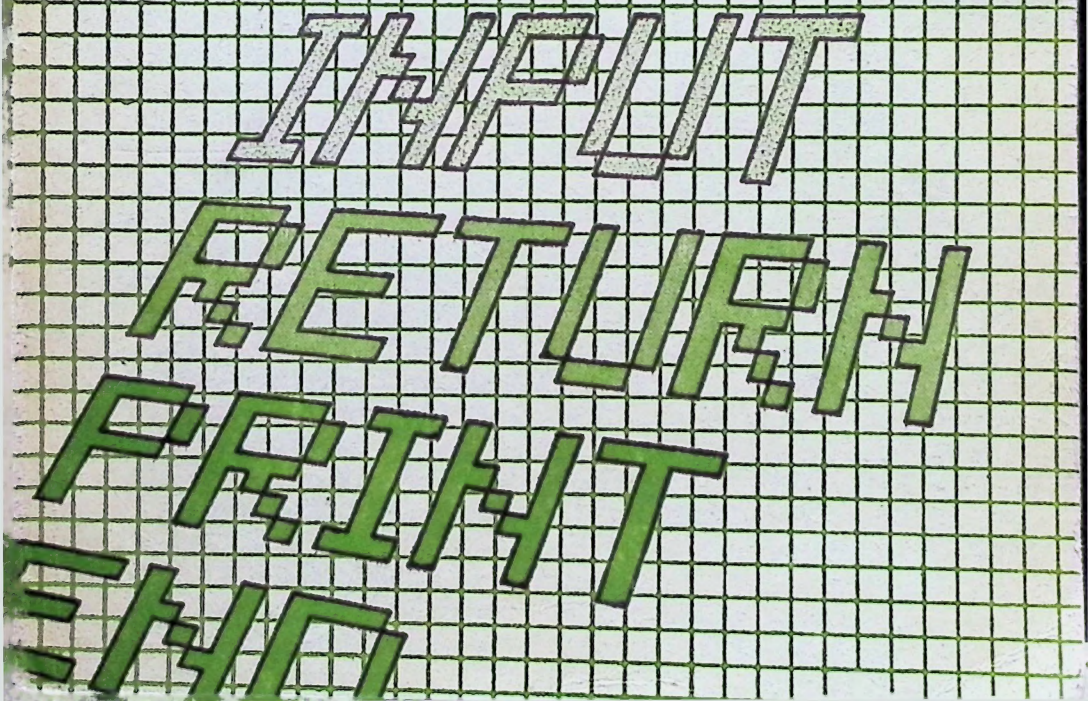
Mikrokomputery

Nauka programowania w języku BASIC dla początkujących

Wacław Iszkowski



Nauka programowania w języku BASIC dla początkujących

Mikrokomputery

Komitet Redakcyjny

Sekretarz WOJCIECH CELLARY
ZUZANNA GRZEJSZCZAK
ANDRZEJ KOBUS
Przewodniczący ROMUALD MARCZYŃSKI
PIOTR MISIUREWICZ
WOJCIECH NOWAKOWSKI
MACIEJ STOLARSKI
HALINA TEMPCZYK
JÓZEF WINKOWSKI
JAN ZABRODZKI

Wacław Iszkowski

Nauka programowania w języku BASIC dla początkujących

Wydawnictwa Naukowo-Techniczne
Warszawa 1987

Opiniodawca *Jan Bielecki*
Redaktor *Ewa Zdanowicz*
Redaktor techniczny *Elżbieta Gontarz*
Okładkę i strony tytułowe projektował *Juliusz Rybicki*

681.3

W książce omówiono podstawowe elementy języka BASIC dla mikrokomputerów. Szczególną uwagę zwrócono na nauczenie prawidłowego programowania. Tekst zilustrowano licznymi przykładami i rysunkami, które zostały przygotowane za pomocą komputera. Przedstawiono również problemy do samodzielnego rozwiązania.
Książka jest przeznaczona dla szerokiego kręgu Czytelników, mających dostęp do popularnych mikrokomputerów.

© Copyright by Wydawnictwa Naukowo-Techniczne,
Warszawa 1987

All rights reserved
Printed in Poland

ISBN 83-204-0834-2

Spis treści

Przedmowa / 9

1. Od algorytmu do programu / 11

1.1. Algorytm / 11
1.2. Zapis algorytmu / 13
1.3. Program / 17
1.4. Struktura programu / 19
Problemy / 22

2. Nauka programowania / 23

2.1. Elementy nauki programowania / 23
2.2. Architektura komputera / 25
2.3. Zapis i wykonanie programu / 28
2.4. Język programowania BASIC / 31
Problemy / 34
Let's learn BASIC English / 35

3. Wprowadzanie tekstu programu / 37

3.1. Początek zabawy / 37
3.2. System BASIC / 40
3.3. Redagowanie wprowadzonego tekstu / 43
3.4. Współpraca z pamięcią kasetową / 47
Styl programowania / 50
Sytuacje błędne / 51
Problemy / 52
Let's learn BASIC English / 53

4. Język Mikro BASIC / 55

4.1. Alfabet języka / 55
4.2. Typy i stałe / 57

4.3. Zmienne / 59
4.4. Instrukcja przypisania / 61
4.5. Instrukcje wprowadzania danych / 63
4.6. Instrukcja wyprowadzania wyników / 66
4.7. Instrukcja komentarza / 70
4.8. Instrukcje zakończenia i wstrzymania / 71
Styl programowania / 72
Sytuacje błędne / 74
Problemy / 76
Let's learn BASIC English / 77

5. Obliczenia numeryczne / 79

5.1. Wyrażenia arytmetyczne / 79
5.2. Standardowe funkcje arytmetyczne / 82
5.3. Wykorzystywanie wyrażeń arytmetycznych / 85
Styl programowania / 89
Sytuacje błędne / 90
Problemy / 92
Let's learn BASIC English / 93

6. Sterowanie obliczeniami / 94

6.1. Wyrażenia relacji / 94
6.2. Instrukcja warunkowa / 96
6.3. Cykliczne wykonywanie instrukcji / 106
6.4. Instrukcje cyklu wyliczanego / 110
6.5. Zagnieżdżone instrukcje cyklu / 115
Styl programowania / 119
Sytuacje błędne / 122
Problemy / 123
Let's learn BASIC English / 127

7. Struktury tablicowe / 129

7.1. Deklarowanie tablic / 129
7.2. Zmienne indeksowane / 132
7.3. Działania na wektorach / 133
7.4. Działania na macierzach / 141
7.5. Tablice trójkątne / 145
7.6. Modyfikacje deklaracji tablic / 147
Styl programowania / 151
Sytuacje błędne / 153
Problemy / 155
Let's learn BASIC English / 157

8. Przetwarzanie tekstów / 158

8.1. Wyrażenia tekstowe / 158
8.2. Standardowe funkcje tekstowe / 163

8.3. Wyrażenia relacji / 172
8.4. Przykład konstruowania programu / 176
Styl programowania / 186
Sytuacje błędne / 187
Problemy / 188
Let's learn BASIC English / 189

9. Funkcje i podprogramy / 190

9.1. Definiowanie i wywoływanie funkcji / 190
9.2. Wykonywanie podprogramów / 194
9.3. Współpraca z podprogramem / 202
Styl programowania / 210
Sytuacje błędne / 213
Problemy / 214
Let's learn BASIC English / 216

10. Uruchamianie programu / 217

10.1. Błędy składniowe i wykonywania / 218
10.2. Błędy logiki programu / 221
10.3. Technika uruchamiania programu / 224
Styl programowania / 229
Let's learn BASIC English / 230

Dodatki / 231

A. Wartości kodów ASCII / 231
B. Wykaz komunikatów błędów / 232
C. Microsoft BASIC-80 / 235
D. Polecenia języka BASIC / 246
E. Spis informacji podstawowych / 247
F. Spis kanonów stylu programowania / 250

Skorowidz / 251

Rodzicom

Przedmowa

Książka ta jest przeznaczona dla wszystkich użytkowników komputerów osobistych. Początkujący mogą poznać w niej podstawy posługiwania się najpopularniejszym obecnie językiem programowania BASIC.

Pojawienie się w latach siedemdziesiątych komputerów osobistych nie wymagających specjalnych warunków eksploatowania, prostych w obsłudze i dość tanich, umożliwiło wielu osobom bezpośredni do nich dostęp. W najprostszy sposób komputer jest wykorzystywany przez dzieci jako urządzenie do gier telewizyjnych.

Komputer osobisty, od prostego ZX SPECTRUM do rozbudowanego IBM PC, nie jest jednak zabawką. Umiejętnie wykorzystany staje się pomocny w wielu dziedzinach życia, oszczędzając czas, zastępując żmudną pracę i wreszcie wprowadzając porządek w gromadzonych i przetwarzanych informacjach. Zakres jego zastosowań jest szeroki, ale też – z czego trzeba sobie zdawać sprawę – ograniczony względami technicznymi.

Książka ta jest więc przeznaczona zarówno dla inżynierów, jak i humanistów czy lekarzy, a także dla studentów i uczniów różnego typu szkół. Taki krąg Czytelników określa oczywiście zakres tematyczny i sposób prezentacji materiału.

Treść książki, po wprowadzeniu podstawowych pojęć z dziedziny informatyki, poświęcona jest przede wszystkim językowi BASIC – jego składni, znaczeniu oraz strukturze tekstu. Najważniejsze zagadnienia z wyróżnionymi grubszą czcionką hasłami są umieszczone na kolorowym tle.

Przykłady prostych i poprawnych programów są integralną częścią wyjaśnień[1]. W tej książce świadomie pominąłem prezentowanie wielu dużych programów, zdając sobie sprawę z tego, że niewielu Czytelników ma tyle cierpliwości i umiejętności, aby czytać i zrozumieć cudze programy. Jednakże najlepiej uczyć się na dobrych przykładach – zachęcam więc do dokładnego ich

[1] Uwaga: Przykłady błędnie zapisanych w języku BASIC fragmentów programów będą oznaczone grubą pionową linią.

przestudiowania. Jeżeli zaś dysponujemy komputerem – warto jest sprawdzić samodzielnie podane rozwiązania, przy czym należy pamiętać, że mogą występować różnice w niektórych konstrukcjach języka BASIC w zależności od typu komputera.

Moim głównym celem było napisanie książki, która nie tyle uczy języka BASIC, ile wyjaśnia zasady poprawnego programowania w tym języku. Stąd też dużo miejsca poświęcono różnego rodzaju typowym i polecanym konstrukcjom struktur programów. Celowi temu służą też podrozdziały zatytułowane *Styl programowania*. Nie należy też opuszczać podrozdziałów *Sytuacje błędne*, w których przedstawiono możliwe przyczyny wystąpienia błędów w uruchamianym programie. Uczenie się na błędach wykrywanych przez system języka BASIC ma szczególnie duże zalety. Dołączone do rozdziałów *Problemy* służą do samodzielnego sprawdzenia swoich umiejętności. Język BASIC (i nie tylko on) jest oparty na elementach języka angielskiego, stąd też w książce znajduje się słowniczek terminów angielskich. Moją radą jest więc nauczyć się terminów na pamięć nawet wtedy, kiedy mamy awersję do nauki języków obcych.

Czytelnicy zaawansowani już w programowaniu mogą ograniczyć się do przeczytania wyróżnionych fragmentów oraz podrozdziałów prezentujących kanony eleganckiego, poprawnego i jedynie dopuszczalnego programowania. Na ten problem zwracam szczególną uwagę, gdyż nauczyć się programować nie jest trudno – znacznie trudniej jest za to oduczyć się programować źle. A źle programować, to znaczy marnować czas swój i komputera, uzyskiwać źle działające programy i mało wiarygodne wyniki ich działania, a w rezultacie zniechęcić się w ogóle do informatyki – czego nikomu nie życzę.

W książce nie wyróżniam żadnego typu komputera osobistego. Posługuję się standardem przemysłowym języka BASIC-80 firmy Microsoft. Omawiam jego ograniczoną wersję, oznaczoną często 2.0. Założeniem moim było przekazanie podstaw nauki programowania w tym języku, na bazie których Czytelnik będzie mógł już samodzielnie oprogramowywać swój komputer osobisty z jego wersją języka BASIC.

Na zakończenie prosimy Czytelników tej książki o nadsyłanie pod adresem Redakcji Elektroniki WNT – z dopiskiem BASIC – swoich uwag na temat prezentacji materiału, o wskazanie, które fragmenty książki są za proste, które za trudne lub niezrozumiałe; prosimy również o podzielenie się swoimi doświadczeniami w poznawaniu tej nowej dziedziny.

Autor wyraża podziękowanie Polskiemu Towarzystwu Informatycznemu za udostępnienie mu mikrokomputera ZX SPECTRUM, na którym zostały wykonane wszystkie rysunki oraz mikrokomputera AMSTRAD CPC 464 do przetestowania programów i wykonania ich wydruków. Bez tej pomocy powstanie tej pracy byłoby niemożliwe.

Warszawa w lutym 1986 r. *Wacław Iszkowski*

1. Od algorytmu do programu

Komputer jest bardzo sprawnym urządzeniem do liczenia i przetwarzania informacji. Wykorzystanie komputera zgodnie z naszymi potrzebami wymaga wstępnego przygotowania go do pracy. Przygotowanie to polega na nauczeniu komputera, jakie ma kolejno wykonać czynności, aby wynik jego działania nas satysfakcjonował. Zanim zajmiemy się omawianiem zasad uczenia komputera, musimy przyswoić sobie następujące twierdzenie:

Komputer sam nie myśli. Efekt działania komputera jest wynikiem naszego myślenia przy formułowaniu dla niego sekwencji zadań do wykonania. Im dokładniej i lepiej przemyślimy sposób rozwiązania danego problemu, tym szybciej i efektywniej otrzymamy wyniki. Jeżeli my popełnimy błąd, to również błędne będzie działanie komputera, uzyskane zaś wyniki nieprawidłowe.

1.1 Algorytm

Na początku mamy ideę rozwiązania pewnego problemu, którego opis jest werbalny i ogólny, nie precyzujący żadnych szczegółów. Dalej w trakcie analizy problemu wyróżniamy w nim pewne charakterystyczne części logicznie od siebie uzależnione. Każdą z tych części dzielimy następnie na mniejsze fragmenty, określając, jakie czynności należy wykonać, aby zrealizować dany fragment. Przy okazji zaczynamy uzmysławiać sobie szczegóły realizacji tych czynności, a także ich możliwe warianty wynikające z wykonania czynności poprzednich. Szczegółowość podziału całego zadania na czynności oraz dokładność określania sposobów wykonania każdej z nich z osobna zależy od umiejętności i możliwości wykonawcy.

Osobie dorosłej wystarczy powiedzieć – włącz radio. Małemu dziecku trzeba czynność tę wyjaśnić dokładniej:
a) Krzysiu podejdź do stolika pod oknem,
b) na stoliku stoi czarna skrzynka z błyszczącymi guziczkami,
c) jeden z guziczków ma czerwoną obwódkę,
d) połóż na tym guziczku paluszek i mocno wciśnij,
e) nie bój się – włączyłeś radio.
Podział danego zadania na poszczególne czynności i dokładny opis ich wykonania nazywamy *algorytmem* realizacji zadania.

Algorytm pochodzenie słowa ze średniowiecznołacińskiego języka na podstawie języka arabskiego

Znaczenia:

1. *cybernetyczne*: dokładny przepis wykonania w określonym porządku skończonej liczby operacji, pozwalający na rozwiązanie każdego zadania danego typu;
2. *matematyczne*: reguła przekształcania wyrażeń matematycznych przez powtarzanie tych samych działań na kolejno otrzymywanych wynikach działań poprzednich.

Źr.: Słownik języka polskiego, PWN 1978, tom I, str. 33.

Do naszych celów definicja cybernetyczna jest użyteczniejsza. W życiu codziennym posługujemy się wieloma algorytmami, nie zdając sobie nawet z tego sprawy. Sposób wyrażania tych algorytmów jest często werbalny, niekiedy mało precyzyjny, pozostawiający wykonawcy możliwość swobodnej interpretacji; ale też, jeżeli popełnimy błąd w przygotowaniu algorytmu, w jego przekazaniu lub w działaniu według niego, skutki mogą być opłakane.

Zanim zajmiemy się algorytmami dla komputerów, przeanalizujmy podany algorytm.

Weź:

4 jaja,
1 szklankę cukru,
1 szklankę mąki,
1 łyżeczkę masła.

Wykonaj:

a) zapal gaz w piekarniku,
b) całe jaja ubij z cukrem,
c) dodaj mąkę – delikatnie mieszając,

d) blachę wysmaruj masłem,
e) wlej masę do blachy,
f) sprawdź, czy temperatura piekarnika osiągnęła 160°C,
g) jeżeli nie – poczekaj,
h) wstaw blachę z ciastem do piekarnika,
i) po wyrośnięciu ciasta zmniejsz gaz,
j) po 15 minutach sprawdź ciasto patyczkiem,
k) jeżeli patyczek jest suchy – wyjmij biszkopt z piekarnika,
l) jeżeli patyczek wilgotny – odczekaj 10 minut i powtórz czynności punktu k.

Wynik:

BISZKOPT. Smacznego.

Spróbujmy teraz czegoś nie dodać lub zmienić kolejność wykonania poszczególnych czynności.

1.2 Zapis algorytmu

Omówię teraz sposób konstruowania algorytmu posługując się przykładem. Na razie bowiem niewiele wiemy o umiejętnościach komputera, a algorytm powinien uwzględniać możliwości wykonawcy.

Wyobraźmy sobie, że chcemy wytapetować mieszkanie. Zanim zakupimy tapety, musimy obliczyć, ile rolek będzie nam potrzeba. Sami będziemy mierzyć ściany, a młodszego brata, chodzącego do drugiej klasy szkoły podstawowej, zatrudnimy do obliczeń. Musimy więc podać mu sposób wykonywania tych obliczeń. Algorytm ten może wyglądać następująco:

Zapis ogólny algorytmu
Dla każdego pokoju zmierzymy powierzchnię ścian i odejmiemy powierzchnię otworów (drzwi i okien). Powierzchnię ścian obliczymy na podstawie podwojonej sumy szerokości i długości pokoju razy wysokość mieszkania (zakładamy, że wszystkie pokoje są prostokątne). Następnie dodamy pewien zapas (10%) na zakładki i ubytki. Policzymy, ile całych brytów (i o jakiej powierzchni) można wykroić z jednej rolki tapety. Obliczymy liczbę rolek i dodamy jedną na resztę powierzchni oraz jedną na zapas.

Zapis szczegółowy algorytmu (dla brata)
1. Zapisz *liczbę_pokoi*
2. Zapisz *wysokość* mieszkania
3. Początkowo potrzebna *powierzchnia_tapet* jest równa 0
4. Dla każdego z pokoi wykonaj (punkty od 5 do 11)
5. Zapisz *szerokość* i *długość* pokoju

6. Oblicz: (*szerokość* + *długość*) * 2 * *wysokość* i wynik zapamiętaj jako *powierzchnię_ścian*
7. Zapisz *szerokość_otworu* i *wysokość_otworu* (drzwi lub okna)
8. Oblicz: *szerokość_otworu* * *wysokość_otworu* i wynik odejmij od *powierzchni_ścian*
9. Spytaj się, czy już wszystkie otwory w pokoju zostały zmierzone?
10. Jeżeli nie, wróć do punktu 7
11. Uzyskaną dotychczas *powierzchnię_ścian* dodaj do *powierzchni_tapet*
12. Odejmij od *liczby_pokoi* jedynkę (ten pokój już zmierzyliśmy) i, jeżeli są jeszcze pokoje do zmierzenia, wróć do punktu 5.
13. Do otrzymanej już *powierzchni_tapet* dodaj 10% na zakładki i ubytki, czyli oblicz:[1]

    powierzchnia_tapet * 0.1 + *powierzchnia_tapet*

    a wynik zapamiętaj jako konieczną do zakupu *powierzchnię_tapet*
14. Zapisz *szerokość_tapety* oraz *liczbę_brytów* równych *wysokości* mieszkania, jakie można z niej wykroić
15. Przyjmiemy, że standardowo *szerokość_tapety* wynosi 0.5 m oraz że można z niej wykroić 2 *bryty*
16. Oblicz *powierzchnię_tapety* według wzoru:

    szerokość_tapety * *liczba_brytów* * *wysokość*

17. Oblicz *liczbę_rolek* tapety jako[2]

    CZĘŚĆ CAŁKOWITA (*powierzchnia_tapet*/*powierzchnia_tapety*) + 1

18. Dodaj jedną rolkę na zapas
19. Podaj, ile *rolek_tapety* według twoich obliczeń trzeba kupić
20. Koniec – dziękuję za pomoc

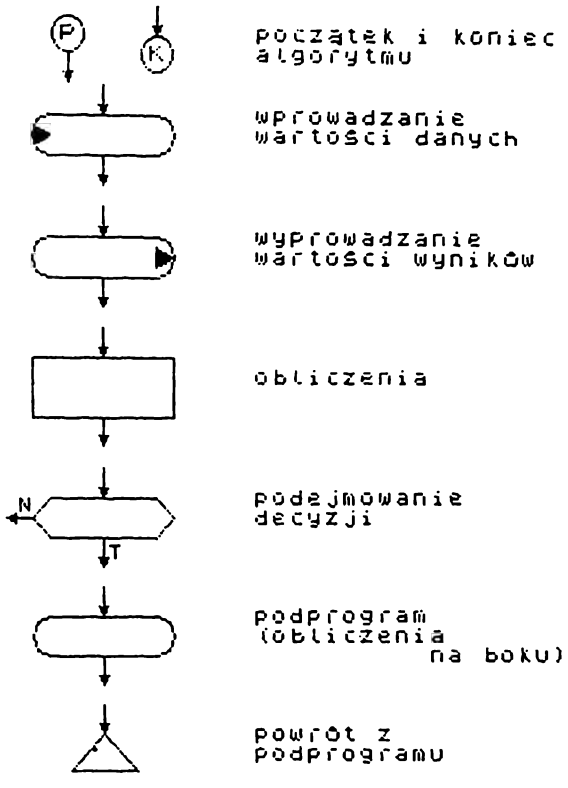
Teraz możemy przystąpić do działania. Bratu dajemy algorytm, papier i ołówek, a sami z centymetrem dokonujemy pomiarów przekazując mu dane do obliczeń. Na końcu oczekujemy na wynik.

Algorytm ten możemy teraz zapisać graficznie w postaci *sieci działań*. Przyjmiemy oznaczenia graficzne pokazane na rys. 1.1.

---

[1] W zapisie liczb ułamkowych do rozdzielenia części całkowitej od dziesiętnej stosujemy znak kropki zamiast przecinka (na wzór krajów anglosaskich).

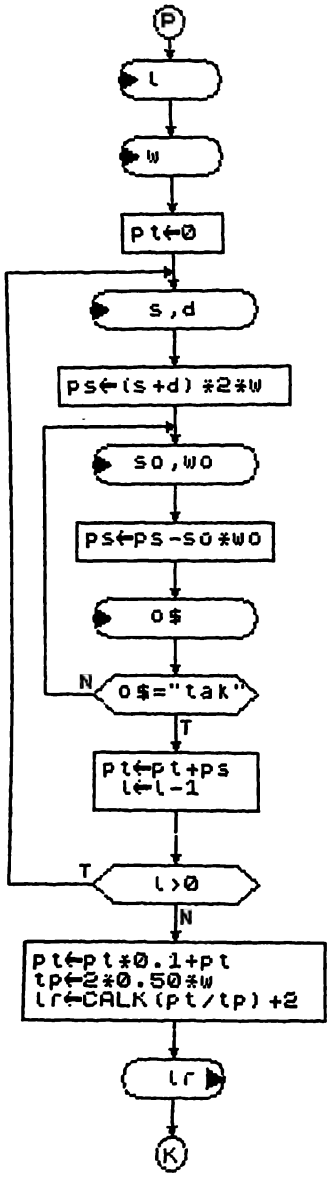
[2] We wzorze dla zaznaczenia dzielenia zamiast kreski ułamkowej używamy kreski ukośnej /, a więc np. $\frac{a}{b}$ dogodniej jest zapisać w jednym wierszu jako *a/b*.



Rys. 1.1

Dla skrócenia zapisu w sieci działań przyjmiemy następujące oznaczenia:

l — *liczba_pokoi*,
w — *wysokość_mieszkania*,
s — *szerokość_pokoju*,
d — *długość_pokoju*,
ps — *powierzchnia_ścian*,
so — *szerokość_otworu*,
wo — *wysokość_otworu*,
o$ — *odpowiedź_na_pytanie_o_wszystkie_otwory* (tak/nie),
pt — *powierzchnia_tapet*,
st — *szerokość_tapety*,
lb — *liczba_brytów*,
tp — *powierzchnia_jednej_tapety*,
lr — *liczba_rolek*.



Rys. 1.2

```
Wczytaj  l
Wczytaj  w
pt←0
Dla każdego  i=1..l
    Wczytaj  s,d
    ps←(s+d)*2*w
        Wczytaj so,wo
        ps←ps-so*wo
        Wczytaj tak lub nie
    Jeżeli nie - to wróć
    pt←pt+ps
pt←pt*0.1+pt
Wczytaj st←0.5,lb←2
tp←st*lb*w
lr←CALK(pt/tp)+1
lr←lr+1
Wyprowadź  lr
```

Rys. 1.3

Zapis opisywanego algorytmu w postaci sieci działań przedstawiono na rys. 1.2. Oprócz sieci działań jest też możliwe stosowanie *diagramów*, które w uproszczony sposób prezentują graficznie algorytm. Dla rozważanego algorytmu odpowiedni diagram pokazano na rys. 1.3. Porównaj obie formy graficznej reprezentacji algorytmu.

Trzeba tutaj zaznaczyć, że również możliwe jest stosowanie form graficznych do ogólniejszego zapisu fragmentów lub całych algorytmów. Zasada ta może w szczególności dotyczyć tych czynności algorytmu, których realizacja jest już dobrze znana i może być w całości wykorzystana.

1.3. Program

Kolejnym krokiem przygotowania komputera do wykonania działań rozwiązujących zadany problem jest zakodowanie algorytmu w postaci *programu.* Zaprojektowany algorytm jest dotychczas zapisany słownie w zrozumiałym dla nas języku. Dla skrócenia zapisu korzystamy ze znanych z algebry zapisów formuł obliczeń, wprowadzając zmienne reprezentujące określone wartości. Dla komputera konieczne jest dodatkowo precyzyjne zapisanie kolejności obliczeń oraz sposobu rozstrzygnięć sytuacji alternatywnych i obliczeń wielokrotnie powtarzanych. Do jednoznacznego wyrażenia tych wszystkich elementów algorytmu służy *język programowania.* Jest to język z jednej strony skonstruowany na podobieństwo języka używanego przez człowieka, a z drugiej — przez dokładne i jednoznaczne określenie reguł zapisu i ich znaczenia jest zrozumiały dla komputera.

Programem jest tekst zapisany w języku programowania opisujący wykonanie sekwencji czynności według zadanego algorytmu. Proces tworzenia programu jest zwany *kodowaniem* algorytmu.

Zanim dokładniej omówimy pojęcie języka programowania oraz podstawowe reguły jego wykorzystywania, powróćmy do naszego algorytmu. Zapiszemy go teraz w postaci programu w języku, który mógłby być językiem programowania (dlaczego nie jest — o tym później).

```
1  WPROWADZ  wartosc  l
2  WPROWADZ  wartosc  w
3  NIECH     pt=0
4  DLA  kazdego  i= od 1  DO  l  wykonaj
5     WPROWADZ  wartosc  s,d
6     OBLICZ    ps=( s+d )* 2 * w
7     WPROWADZ  wartosc  so,wo
8     OBLICZ    ps=ps - so*wo
9     WYSWIETL  tekst
      "Czy zmierzyles juz wsystkie otwory(TAK/NIE)?"
      WPROWADZ  tekst  o$
10    JEZELI    o$= jest rowne tekstowi "NIE" WTEDY  WROC DO 7
11    OBLICZ    pt=pt+ps
12    KOMENTARZ wartosc  i  zwiekszamy o 1 i sprawdzamy
         czy nie juz wieksza od wartosci  l,jezeli nie to
   NASTEPNE  i
13 OBLICZ   pt=pt*0.1 + pt
14 WCZYTAJ  wartosci  st,lb  z zestawu danych stalych
15 DANE     0.5, 2
16 OBLICZ   tp=st * lb * w
17 OBLICZ   lr=CZESC CALKOWITA ( pt/tp )+1
18 OBLICZ   lr=lr+1
19 WYPROWADZ  tekst "liczba rolek tapety = "; i wartosc lr
20 KONIEC
```

Musisz teraz Czytelniku przyznać, że zapis tego programu jest całkiem zrozumiały i właściwie nie trzeba nic dodać nic ująć. Dlaczego więc tekstu tego nie rozumie żaden z dostępnych komputerów? Ojczyzną komputerów są Stany Zjednoczone i tam w 1946 r. narodził się pierwszy komputer ENIAC, którego kolejni potomkowie zostali nauczeni języków zbliżonych do języka angielskiego. Jednym spośród wielu używanych języków programowania jest język BASIC. Zapiszemy więc również i nasz program w tym języku, po prostu tłumacząc go na język angielski i odrzucając zbędne słowa. Czytelnicy znający język angielski nie będą mieli tutaj kłopotu w zrozumieniu tego tłumaczenia.

```
1 INPUT  l
2 INPUT  w
3 LET  pt=0
4 FOR i=1  TO  l
5   INPUT  s,d
6   LET  ps= (s+d)*2*w
7   INPUT  so,wo
8   LET  ps= ps- so*wo
9   INPUT  "Czy wszystkie otwory(TAK/NIE)"; o$
10  IF  o$="NIE"  THEN GOTO 7
11  LET  pt=pt+ps
12 NEXT i
13 LET  pt=pt*0.1+pt
14 READ st,lb
15 DATA .5,2
16 LET  tp=st*lb*w
17 LET  lr=INT(pt/tp)+1
18 LET  lr=lr+1
19 PRINT  "liczba rolek tapety =";lr
20 END
```

Otrzymaliśmy gotowy do wykonania program w języku BASIC.

1.4. Struktura programu

W programie możemy wyróżnić kilka charakterystycznych *elementów składowych*. Postać ich zapisu jest zależna od przyjętych konwencji w języku programowania.

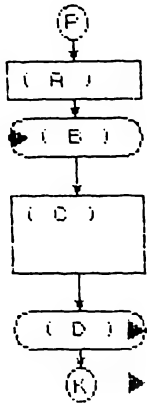
Elementami składowymi programu są:
1. *Stałe* — wartości wpisane bezpośrednio w tekst programu.
2. *Zmienne* — reprezentujące pojedynczą wartość, zmienianą w wyniku wykonywania obliczeń.
3. *Dane* — wartości wprowadzane z otoczenia programu.
4. *Wyniki* — wartości wyprowadzone ze zmiennych z programu w jego otoczenie.
5. *Instrukcje* — operacje określające wykonanie lub decyzję o wykonaniu jednostkowego obliczenia.

W naszym programie elementami składowymi są:

1. Stałe: 0 w linii numer 3; 1 w linii numer 4, 17, 18; 0.5 w linii numer 13; 0.5 i 2 w linii numer 15.
2. Zmienne: *l*, *w*, *pt*, *i*, *s*, *d*, *ps*, *so*, *wo*, *oS*, *st*, *lb*, *tp*, *lr*.
3. Dane: wartości dla zmiennych: *l*, *w*, *s*, *d*, *so*, *wo*, *oS*; wartość 0.5 dla *st* i wartość 2 dla *lb*.
4. Wyniki: wartość zmiennej *lr*.
5. Instrukcje: INPUT, LET, FOR...TO, IF...THEN, GOTO, NEXT, READ, DATA, PRINT, END.

Zapis algorytmu w postaci sieci działań prezentuje graficznie przyszłą strukturę programu. Omawiając więc podstawowe struktury programów będziemy posługiwać się sieciami działań. Należy przy tym zaznaczyć, że dwuwymiarowość sieci działań musi być w trakcie kodowania sprowadzona do sekwencji linii tekstu zawierających instrukcje języka.

Moduły programu. Program funkcjonalnie jest złożony z czterech głównych części (modułów).
A. Moduł deklaracji zmiennych z nadawaniem im wartości początkowych.
B. Moduł wprowadzania wartości danych.
C. Moduł wykonywania obliczeń zasadniczych.
D. Moduł wyprowadzania wartości wyników.



Rys. 1.4

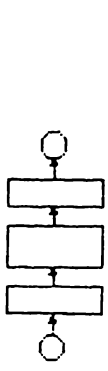
Graficznie podział programu na moduły przedstawia rys. 1.4. W praktyce wymienione moduły mogą występować wielokrotnie, w różnych wzajemnych powiązaniach. W naszym programie przypisanie rodzaju modułu każdej z linii jest następujące:

linie programu:	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
rodzaj modułu:	B	B	A	A	B	C	B	C	B	C	C	C	C	B	A	C	C	C	D	

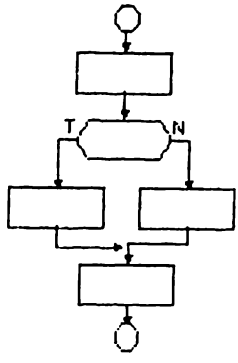
Struktura programu. Moduły podziału funkcjonalnego oraz części tych modułów tworzą jedną z czterech podstawowych struktur programów:

1. *Program o strukturze liniowej*, w którym kolejne instrukcje tworzą sekwencję obliczeń (rys. 1.5).
2. *Program o strukturze rozgałęzionej* (decyzyjnej), w którym w każdej z części niektóre z obliczeń są wykonywane przy spełnieniu obliczonych wartości warunków, a inne w przypadku ich niespełnienia (rys. 1.6).
3. *Program o strukturze cyklicznej* (iteracyjnej), w którym część instrukcji jest wielokrotnie wykonywana aż do spełnienia określonego warunku (rys. 1.7).
4. *Program o strukturze proceduralnej*, w którym wyróżnione części programu (zwane podprogramami) są logicznie włączone w wykonanie różnych innych części programu (rys. 1.8).

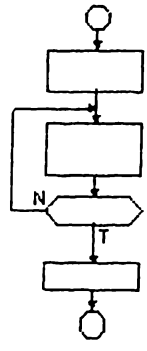
Praktycznie biorąc w każdym programie można znaleźć każdy z rodzajów struktur w zawężeniu do jego modułów. Wyróżnienie tych struktur miało na celu wyjaśnienie pierwszych podstawowych pojęć oraz zwrócenie uwagi Czytelnika na



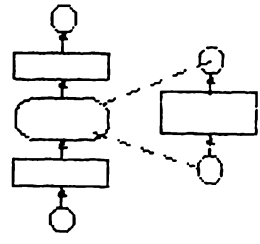
Rys. 1.5



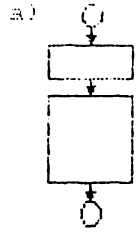
Rys. 1.6

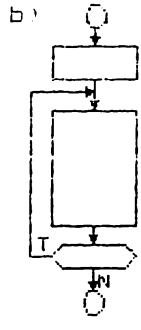


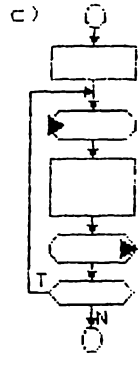
Rys. 1.7

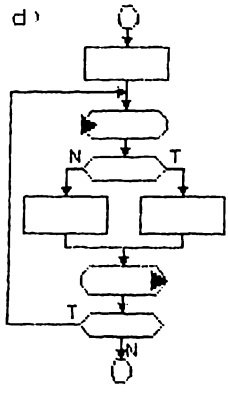


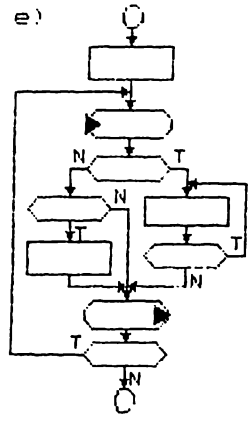
Rys. 1.8











Rys. 1.9

konieczność zachowania ładu w projektowanym algorytmie, a potem w zakodowanym programie. Jeżeli bowiem na danym stopniu szczegółowości opisu całego lub części algorytmu możemy przyporządkować mu odpowiednią strukturę, to mamy dużą gwarancję, że nasz program będzie dobry. Ilustracją tych rozważań jest rys. 1.9, prezentujący kolejne uszczegółowienie algorytmu z zachowaniem możliwości lokalnego określenia typu struktury.

Problemy

Skonstruuj algorytm i zapisz go słownie oraz graficznie w postaci sieci działań lub diagramu dla następujących problemów:

1.1. Poszukaj pierwiastków równania kwadratowego.

Pamiętaj o osobliwościach rozwiązania. Jest to klasyczny problem, od którego rozpoczynają się prawie wszystkie podręczniki z zakresu nauki programowania – prosty, ale wystarczająco skomplikowany.

1.2. Niech (x, y) oznaczają współrzędne punktu na płaszczyźnie:

a) sprawdź, czy prosta przechodząca przez dwa punkty $(x_1, y_1), (x_2, y_2)$ jest równoległa do prostej $y = ax + b$;

b) sprawdź, czy przez punkty $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ można przeprowadzić jedną prostą;

c) określ, jakiego typu jest trójkąt wyznaczony przez punkty (x_1, y_1), (x_2, y_2), (x_3, y_3) – równoboczny, równoramienny czy prostokątny.

1.3. Z otoczenia programu jest wprowadzany ciąg liczbowy z ostatnim wyrazem równym zeru:

a) sprawdź, czy jest to ciąg o wartościach rosnących (z wyjątkiem wartości ostatniego wyrazu);

b) oblicz sumę wartości i oblicz średnią wartość ciągu;

c) sprawdź, czy wyrazy ciągu tworzą postęp arytmetyczny.

1.4. Korzystając ze wzoru iteracyjnego na poszukiwanie pierwiastka funkcji

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

w którym: x_n – kolejne przybliżenie wartości pierwiastka; $f(x_n)$ – wartość funkcji dla wartości x_n; $f'(x_n)$ – wartość pierwszej pochodnej dla wartości x_n; x_{n+1} – nowe przybliżenie wartości pierwiastka, dopóty, dopóki $|x_{n+1} - x_n| > \varepsilon$; oblicz:

a) pierwiastek funkcji $f(x) = x^2 - a$

b) pierwiastek funkcji $f(x) = x^2 - x + \ln x$

c) pierwiastek funkcji $f(x) = ax^2 + bx + c$

sprawdzając, czy proces nie jest rozbieżny.

2. Nauka programowania

Czas już wyjaśnić poszczególne słowa tytułu tej książki. Słowo „nauka" jest ogólnie zrozumiałe. Pojęcie programowania komputerów zostało przybliżone Czytelnikowi w poprzednim rozdziale.

Programowanie komputera jest zespołem czynności umożliwiających rozwiązanie zadanego problemu i obejmuje: ustalenie algorytmu, określenie sposobu i kontroli wprowadzania danych i wyprowadzania wyników, zapisanie algorytmu w języku zrozumiałym dla komputera oraz weryfikację poprawności działania programu.

Następnie musimy dokładniej sprecyzować pojęcie języka programowania, a w szczególności wyróżnić język BASIC. I wreszcie dla zrozumienia zasad programowania oraz sposobu wykonywania programu należy przynajmniej w skrócie omówić zasady funkcjonowania komputera.

2.1. Elementy nauki programowania

Definicja pojęcia programowania nie wyjaśnia, jakie są kryteria oceny efektu programowania i jak powinien prawidłowo przebiegać ten proces. Wynikiem programowania jest zapis pewnego algorytmu w pewnym języku programowania. Niestety sam proces programowania nie jest w pełni algorytmiczny. Oprócz wielu zasad opracowanych na podstawie doświadczeń tysięcy programistów, w procesie programowania jest niezbędne logiczne i analityczne myślenie programującego, a pożądane też trochę talentu i zdolności. Tym ostatnim aspektem będziemy się zajmować przy dyskusji stylu programowania. Tutaj omówimy pokrótce tylko podstawowe metody i zasady programowania. Pełniejsze zrozumienie tych zagadnień wymaga bowiem już pewnych umiejętności i doświadczeń.

Istnieje kilka metod konstruowania algorytmu, a na jego podstawie — programu. Dalej będziemy utożsamiali konstruowanie algorytmu i jego zapis w postaci programu. Bardziej doświadczeni programiści dla mniej skomplikowanych algorytmicznie problemów mogą bowiem od razu zapisywać rozwiązanie w postaci tekstu programu.

Metody konstruowania programu:

1. *Od ogółu do szczegółu* — czyli zadany problem dzielimy na logicznie domknięte moduły posługując się ogólnym, werbalnym opisem ich funkcjonowania. Następnie każdą z tych części dzielimy na coraz mniejsze moduły, uzyskując w rezultacie opis pojedynczych czynności w instrukcjach języka.
2. *Od szczegółu do ogółu* — czyli rozpoczynamy rozwiązanie problemu od skonstruowania obliczeń dla jego najistotniejszych fragmentów, koncentrując się na uzyskaniu gotowego zapisu. Następnie fragment ten lub też inne podobnie uzyskane fragmenty łączymy ze sobą dołączając instrukcje wprowadzania danych, wyprowadzania wyników oraz inne instrukcje pomocnicze.

Podany opis metod nie jest w pełni precyzyjny. W praktyce też z reguły łączymy obie metody, gdyż często od znalezionej idei rozwiązania podstawowej funkcji programu zależy pozostała jego część. Warto również pamiętać, że w procesie programowania uszczegółowienie programu jest często wynikiem procesu jego uruchamiania, w trakcie którego zostają wykryte błędy w funkcjonowaniu. Zawsze też powinniśmy uzmysłowić sobie główny cel napisania danego programu, unikając zbytnich — w gruncie rzeczy niepotrzebnych — ozdobników.

W samej nauce programowania wyróżniamy:

- konstruowanie algorytmów polegające na logicznym podziale zadanego problemu na pojedyncze czynności;
- poznawanie i rozumienie algorytmów już istniejących oraz ich wykorzystywanie,
- weryfikowanie poprawności algorytmu i jego zgodności z zadanym problemem,
- poznawanie instrukcji języka — ich wariantów i ograniczeń,
- stosowanie standardowych rozwiązań niektórych modułów programu,
- wykorzystywanie funkcji dostarczonych standardowo z danym językiem programowania,
- adaptowanie rozwiązań już istniejących,
- efektywne wykorzystywanie funkcji systemu cyfrowego, jakim jest komputer i jego firmowe oprogramowanie,
- uruchamianie gotowego programu polegające na umiejętności wykrywania i usuwania błędów przy korzystaniu z programów wspomagających,

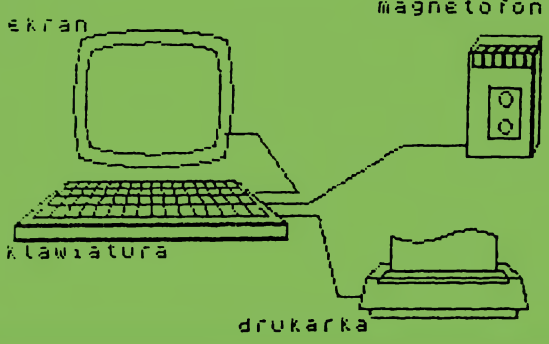
- ocenę efektywności działania programu,
- weryfikowanie poprawności programu,
- prawidłową eksploatację gotowego programu,
- dokumentowanie oraz rzetelne i zrozumiałe opisywanie zasad korzystania z programu.

Łatwo zauważamy, że nauka programowania nie jest sztuką prostą. Wymaga ona poznania wielu nowych elementów. Co więcej, elementy te wymagają często zmiany lub przyswojenia sobie nowych schematów myślenia. Musimy zacząć myśleć logicznie, sprawnie, a nasze myśli muszą być wyrażone bardzo precyzyjnie i jednoznacznie. W naszym myśleniu i działaniu musi być też zachowany porządek. Wszelki chaos i bałagan skazuje naszą pracę na niepowodzenie lub długotrwałe, uciążliwe i wreszcie zniechęcające nas postępowanie. Wynik programowania jest bowiem weryfikowany przez działanie komputera, a ten, jak nikt i nic innego, akceptuje tylko i wyłącznie porządek.

2.2 Architektura komputera

Nic jest moim celem wyjaśnianie tutaj zasad działania komputera. Jednak dla łatwiejszego zrozumienia wyłożonych dalej zagadnień konieczne jest poznanie przynajmniej elementarnych pojęć z tym związanych. Przy omawianiu zasad konstruowania algorytmów, powiedzieliśmy też, że konieczna jest znajomość możliwości wykonawcy – w tym przypadku komputera. Tutaj w skrócie zaprezentujemy architekturę najprostszej konfiguracji komputera, zwanego komputerem osobistym lub mikrokomputerem.

Komputer osobisty jest złożony z *procesora*, *pamięci operacyjnej* oraz dołączonych do niego *urządzeń zewnętrznych* takich, jak: klawiatura, ekran odbiornika telewizyjnego lub monitora, magnetofon kasetowy, głośnik oraz ewentualnie drukarka (rys. 2.1).



Rys. 2.1

Omawianie cech funkcjonalnych poszczególnych składników komputera rozpoczniemy od prezentacji urządzeń zewnętrznych, mających dotychczas nieco inne przeznaczenie.

Klawiatura jest urządzeniem do wprowadzania informacji do komputera w postaci sekwencji znaków. Zawiera ona w standardowym układzie QWERT – znanym z maszyn do pisania – klawisze znaków liter (niekiedy tylko dużych), cyfr, znaków specjalnych (typu +, −, %, * itp.), sterujących (typu SHIFT, RETURN itp.) oraz funkcyjnych. Naciśnięcie danego klawisza powoduje wygenerowanie kodu danego znaku i przyjęcie go przez procesor pod warunkiem, że procesor jest na to przygotowany. W przeciwnym razie znaki te są gubione.

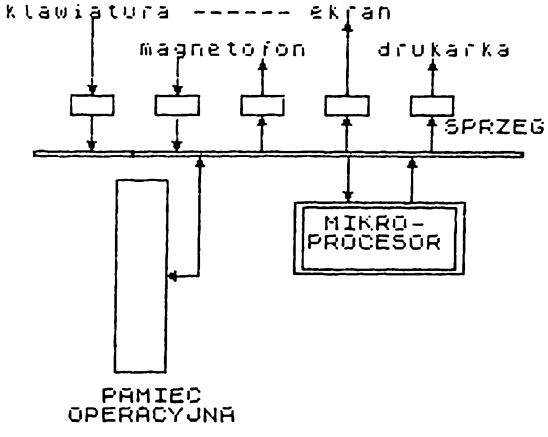
Ekran telewizora lub specjalnego *monitora* jest urządzeniem do wyświetlania znaków. Umożliwia on jednoczesne wyświetlenie kilkunastu (od 16 do 32 dla różnych rodzajów monitorów) *wierszy* tekstu złożonego z kilkudziesięciu znaków (od 24 do 132). Kolejne pozycje znaków w wierszu są ponumerowane umownie, a pozycje o identycznych numerach we wszystkich wierszach tworzą *kolumny*. W zasadzie wszystkie znaki (oprócz sterujących i funkcyjnych) wprowadzane z klawiatury są wyświetlane bezpośrednio na ekranie. Po zapełnieniu ekranu nowy wolny wiersz powstaje przez przesunięcie wszystkich wierszy o jeden wiersz w górę. Pierwszy wiersz zostaje wtedy usunięty z ekranu. Dla potrzeb graficznych ekran może być też traktowany jak tablica złożona z pewnej liczby *punktów* uporządkowanych również w wiersze i kolumny. W wersji dwubarwnej ekranu (czarno-białej) każdy punkt może być koloru tła (ciemny) lub jasny (biały). W wersji kolorowej ekranu każdy punkt może przyjmować jeden z kilku lub kilkunastu kolorów.

Magnetofon kasetowy jest urządzeniem do przechowywania (magazynowania) informacji na kasecie z taśmą magnetyczną. Urządzenie to jest nazywane *pamięcią pomocniczą*. Informacja na taśmie jest analogowym zapisem informacji cyfrowej. Proszę sprawdzić, co będziemy słyszeć, gdy kasetę z programem będziemy odtwarzać jako nagranie grupy Manaam. Zawartość pamięci pomocniczej może być odczytywana i zapisywana bezpośrednio przez komputer.

Głośnik, wbudowany w komputer, jest przeznaczony głównie do sygnalizacji dźwiękowej o stanie programu i systemu. Wysokość i długość dźwięków sygnału jest definiowana cyfrowo przez aktualnie działający program.

Drukarka jest urządzeniem do wyprowadzania informacji. Zasada jej działania może być różna – od typowej dla maszyny do pisania, poprzez wersje mozaikowe i termiczne, aż do laserowej. Nie wdając się w szczegóły techniczne, należy pamiętać, że po wyprowadzeniu pojedynczego wiersza tekstu, papier jest wysuwany o jeden wiersz w górę i do poprzedniego wiersza nie ma już w zasadzie powrotu. Istnieją też drukarki umożliwiające wydrukowanie rysunków.

Teraz rodzi się pytanie — a gdzie jest ten procesor? Na pewno nie ma go w telewizorze, magnetofonie i pewnie też nie ma go w drukarce. Pozostaje więc tylko klawiatura. W pudełku z klawiaturą (czasem jednak również w osobnym pudełku) znajduje się płytka z układami scalonymi. Jeden z tych układów jest procesorem, inne blokami pamięci operacyjnej, a pozostałe układami sprzęgającymi z urządzeniami zewnętrznymi (rys. 2.2).



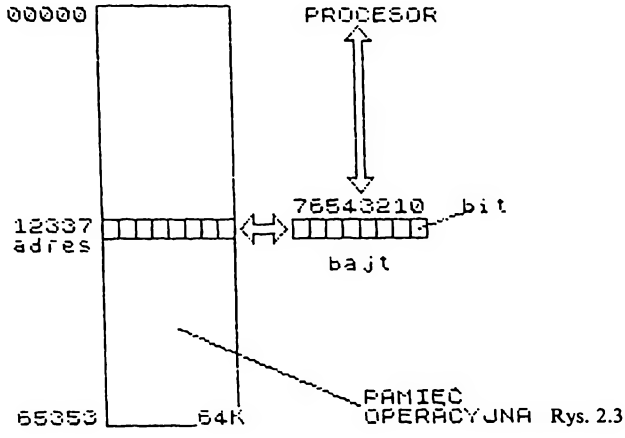
Rys. 2.2

Centralna jednostka przetwarzająca — procesor, a dokładniej *mikroprocesor* — jest podstawowym elementem komputera wykonującym kolejno pobierane z pamięci operacyjnej rozkazy programu. W zestawie rozkazów istnieją rozkazy operacji arytmetycznych i logicznych na argumentach pobranych z pamięci operacyjnej lub umieszczonych w rejestrach roboczych procesora. Istnieją również rozkazy bezwarunkowej lub warunkowej zmiany sekwencji pobrania rozkazów oraz rozkazy współpracy z pamięcią operacyjną i urządzeniami zewnętrznymi. W procesorze znajdują się jednostki arytmetyczno-logiczne realizujące cztery podstawowe działania.

Pamięć operacyjna (PaO) służy do przechowywania kodu rozkazów i adresów ich argumentów oraz wartości. Pojedynczym elementem pamięci jest bajt złożony z 8 bitów. Pojedynczy bit umożliwia zapamiętanie dwóch wartości: 0 lub 1. Bajty pamięci są uporządkowane w kolejności od 0. Numer bajtu w pamięci jest nazywany adresem bajtu (rys. 2.3). Jednostką wielkości pamięci jest kilobajt (KB) równy 1024 bajtom. Najczęściej spotykanymi rozmiarami pamięci operacyjnej są:

4K, 8K, 16K, 32K, 48K i 64K bajtów.

Pamięć operacyjna w komputerach osobistych jest półprzewodnikowa.



Rys. 2.3

Dzieli się ona na *pamięć stałą* (ROM) o stałej (niezmiennej) zawartości oraz *pamięć zmienną* (RAM), której zawartość może być modyfikowana podczas pracy procesora.

Pamięć zmienna (RAM) jest ulotna, gdyż jej aktualna zawartość po wyłączeniu zasilania lub po wyzerowaniu komputera (wykonanie RESET) jest niszczona.

Pamięć stała (ROM) przechowuje oprogramowanie stałe komputera – system zarządzająco-operacyjny oraz często program interpretatora systemu BASIC.

Komputery osobiste (dla których prezentujemy tutaj język BASIC) są składane z 8-bitowych[1)] mikroprocesorów typu Intel 8080, Z80 lub ich pochodnych oraz z bloku pamięci operacyjnej o wielkości do 64K bajtów (częściowo zawierających pamięć stałą) i ze sprzęgów z typowymi urządzeniami zewnętrznymi (klawiatura, ekran, głośnik, magnetofon).

[1)] Cecha 8 bitów określa właściwość konstrukcyjną procesora umożliwiającą w jednym cyklu przetworzyć wartość zapisaną w jednym bajcie (8 bitach).

2.3. Zapis i wykonanie programu

Komputer posługuje się informacjami zakodowanymi w postaci sekwencji dwóch wartości 0 i 1. Również rozkazy są w ten sposób zakodowane, a więc program wykonywany w komputerze musi być zapisany w kodzie (języku) wewnętrznym.

Zewnętrzna, czytelna dla programisty, postać kodu wewnętrznego jest nazywana *językiem asemblerowym* (niskiego poziomu). W języku tym każdy rozkaz jest opisany mnemonicznym hasłem instrukcji, a większość odwołań do argumentów (ich adresów) jest zadawana symbolicznie.

Zapisywanie treści programu w języku asemblerowym jest dość uciążliwe i czasochłonne. Z czasem powstały więc języki programowania wysokiego poziomu, bardziej zrozumiałe dla programującego. Języki te, jak już było powiedziane, zostały głównie oparte na słownictwie angielskim oraz regułach zapisu wyrażeń z algebry. W przeciągu dość krótkiego czasu życia informatyki opracowano na świecie ponad 1000 różnych języków programowania. Cechami rozróżniającymi je są: odmienność sposobu zapisu niektórych konstrukcji algorytmów, uproszczenia zapisu konstrukcji typowych, podział na moduły oraz różny stopień wspomagania programisty w jego działaniach.

Obecnie, na szczęście, tylko kilka spośród tych języków zyskało uznanie. Najpopularniejszymi językami są:

PL/I, Cobol, Fortran, BASIC, Pascal, C, Forth, Modula-2, Ada.

W opisie języka programowania wyróżniamy opis składni i semantyki.

Składnia języka jest zbiorem reguł określających prawidłowe postacie zapisu treści programu. Inaczej mówiąc, są to reguły ortografii i gramatyki języka specyfikujące poprawne postacie deklaracji instrukcji i programu.

Semantyka języka jest zestawem znaczeń określających jednoznacznie sposób interpretacji poszczególnych instrukcji języka. W przypadku języka programowania semantyka precyzuje sposób zrozumienia każdego z elementów programu przez komputer.

Wiemy już, że funkcjonowanie komputera jest wynikiem wykonywania przez procesor kolejnych rozkazów pobieranych z pamięci operacyjnej. Nasuwa się więc pytanie — w jaki sposób można uzyskać sekwencję rozkazów dla każdej z instrukcji języka wysokiego poziomu?

Translator jest częścią oprogramowania komputera i służy do przekształcenia źródłowego tekstu programu, zapisanego w określonym języku programowania, w równoważny funkcjonalnie program wykonywany bezpośrednio przez procesor.

Wyróżniamy dwie podstawowe, odmienne zasady funkcjonowania translatora:

1. *Interpretator* jest translatorem zapisującym źródłowy tekst programu bezpośrednio w pamięci operacyjnej, a następnie wykonującym program poprzez interpretację semantyczną każdej zapisanej instrukcji wywołując odpowiednią procedurę.

2. *Kompilator* jest translatorem tłumaczącym postać źródłową programu w postać wynikową jako sekwencję rozkazów zgodnie ze znaczeniem semantycznym. Po wykonaniu kompilacji, gotowy, sprawdzony pod względem poprawności składni, program może być wielokrotnie wykonywany.

Dla wyjaśnienia różnicy między interpretatorem a kompilatorem wyobraźmy sobie, że mamy tekst instrukcji obsługi pewnego urządzenia – zapisany w obcym języku, którego w ogóle nie znamy – na przykład po japońsku. Jeżeli chcemy teraz korzystać z tej instrukcji, możemy wybrać dwie drogi postępowania.

Pierwsza metoda polega na zatrudnieniu osoby znającej język japoński. Osoba ta odczytuje i tłumaczy kolejne punkty instrukcji, które po kolei realizujemy. Istotne jest tutaj, że kolejność wybierania tych punktów jest uwarunkowana aktualnym stanem urządzenia. Trzeba też dodać, że interpretowanie tych samych punktów instrukcji przez tę osobę może odbywać się wielokrotnie.

Druga metoda polega na pełnym przetłumaczeniu przez tłumacza instrukcji na nasz język ojczysty, po czym wersję oryginalną instrukcji chowamy, a sami posługujemy się wielokrotnie tłumaczeniem.

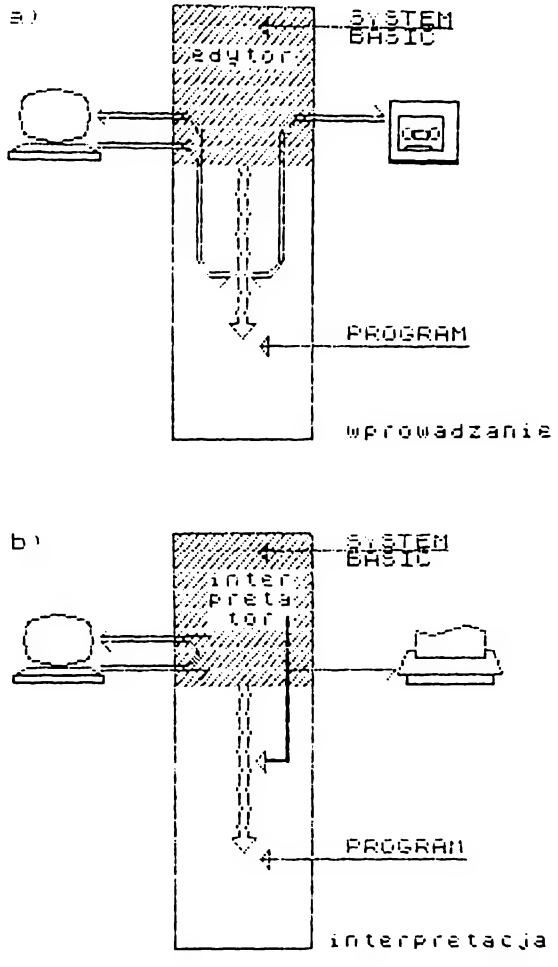
Pierwsza metoda wydaje się być mniej atrakcyjna, bo w każdej chwili musimy mieć do dyspozycji osobę tłumacza – interpretatora oryginalnego tekstu instrukcji. Jeżeli jednak założymy, że kontrahent zagraniczny będzie jeszcze na bieżąco poprawiał instrukcję, to pełne jej tłumaczenie ma sens dopiero po zakończeniu etapu modyfikacji.

W komputerach osobistych przeważnie korzysta się z interpretatora. Zanim jednak zaprezentujemy schemat funkcjonowania systemu – komputer i interpretator – podsumujmy dotychczas poznane elementy nauki programowania.

Fazy życia programu:

1. Idea rozwiązania problemu przy użyciu komputera.
2. Zaprojektowanie algorytmu.
3. Zapisanie algorytmu w języku programowania jako programu.
4. Wprowadzenie treści programu do pamięci operacyjnej.
5. Translacja programu.
6. Wykonanie programu z wprowadzeniem danych i wyprowadzeniem wyników.

Podane uproszczone fazy życia programu nie wyczerpują wszystkich niuansów. Przy korzystaniu do translacji z interpretatora, trzy ostatnie fazy są sterowane jednym programem zarządzającym – nazywanym dla języka BASIC – *systemem* BASIC. System ten na stałe znajduje się w pamięci pomocniczej i oprócz interpretatora zawiera również program edytora umożliwiający wprowadzenie



Rys. 2.4

treści naszego programu. Oba te programy mogą działać na przemian, to zmieniając treść programu, to interpretując ją dla zadanych danych źródłowych, przekazując wyniki działania wykonywanego przez nas programu. Schemat takiego działania systemu BASIC przedstawiono na rys. 2.4.

2.4. Język programowania **BASIC**

Po omówieniu cech charakterystycznych języków programowania wysokiego poziomu, osobny rozdział należy poświęcić językowi BASIC.

Język BASIC został zaprojektowany przez profesorów Johna G. Kemeny'ego i Thomasa F. Kurtza z Uniwersytetu w Darthmouth (USA) w 1964 roku. Nazwa języka jest akronimem hasła:

Beginner's All purpose Symbolic Instruction Code

(Kod instrukcji symbolicznych do wszystkich celów dla początkujących).

Minęło więc ponad 20 lat od chwili powstania języka. W informatyce jest to już wiek balzakowski. Warto jednak prześledzić bogatą w wydarzenia historię kształtowania się tego języka i jego kolejne losy.

Język został zaprojektowany głównie dla własnych potrzeb prostego zapisu programów obliczeń arytmetycznych. Cechą charakterystyczną już pierwszej wersji języka było numerowanie linii, proste instrukcje wprowadzania danych i wyprowadzania wyników, brak jawnego deklarowania zmiennych oraz operacje na tablicach (dodawanie i mnożenie tablic było zapisywane jedną operacją).

Ku zdumieniu autorów język zaczął zyskiwać popularność i kolejne znane firmy komputerowe zaczęły dostarczać interpretatory lub kompilatory języka BASIC. W krótkim czasie BASIC był rozpowszechniony w dużych systemach o dostępie bezpośrednim działających na komputerach: IBM, ICL, PDP-8, PDP-11, CDC, HP, itd. W trakcie implementacji, podstawowa wersja języka BASIC była rozszerzana o nowe konstrukcje. Wprowadzono wartości całkowite i napisy oraz operacje przetwarzania tekstu. Wzbogacono też instrukcje wejścia-wyjścia o współpracę z dużymi systemami plików istniejącymi w tych systemach (BASIC Plus). Niestety brak ogólnie uznanego standardu – czyli normy języka – spowodował, że te rozszerzenia języka BASIC znacznie różniły się od siebie pod względem składni zapisu i semantyki, co nie ułatwiało przenoszenia programów między różnymi instalacjami.

Tak upłynęło 10 lat. W 1974 roku drogi rozwoju języka rozeszły się. Jedna z nich rozpoczęła się od zaimplementowania języka dla mikroprocesora Intel. Ta wersja języka była bardzo ograniczona i wymagała jedynie 4K pamięci dla programu interpretatora. W tym też czasie powstał pierwszy mikrokomputer Altair 8800. Firma MITS produkująca ten mikrokomputer nabyła licencję na opracowany interpretator języka i w 1975 roku narodziła się firma Microsoft. Wykorzystywany język BASIC został nazwany Microsoft BASIC (MBASIC) wersja 1.0. BASIC rozpoczął karierę mikrokomputerową.

W 1976 roku powstał kolejny MBASIC wersja 2.0 wymagająca już 8K pamięci. Licencję na tę wersję języka nabył Commodore dla swoich komputerów osobistych, rozszerzając ją następnie według własnych projektów.

W 1978 roku Microsoft udzielił licencji firmie TANDY Corporation produkującej TRS-80 Model I. Kolejne rozszerzenia języka, przypominające coraz bardziej najpełniejsze postacie zaimplementowane uprzednio na dużych komputerach nosiły nazwy Level I BASIC oraz Level II BASIC (MBASIC wersja 2.0) oraz wreszcie Model III BASIC czyli Extended BASIC. Interpretator tej ostatniej wersji zajmował już 12K pamięci i zawierał między innymi elementy grafiki oraz generacji dźwięku. W firmie Microsoft był nazywany MBASIC wersja 3.0. W 1979 roku dołączono do języka BASIC operacje współpracy z plikami na dyskach i dyskietkach. Powstała wersja DISK BASIC zwana Microsoft BASIC (MBASIC) wersja 4.0

Również inne firmy produkujące komputery osobiste mniej lub bardziej legalnie adaptowały Microsoft BASIC dla swoich potrzeb. Sprzęt ten wykorzystywał głównie 8-bitowy procesor Intel 8080, Z80 lub podobne, stąd też oprogramowanie w uzupełnieniu nazwy miało oznaczenie 80. Z braku standardu opracowanie firmy Microsoft stało się standardem przemysłowym postaci języka BASIC-80 (DISK BASIC, Microsoft BASIC wersja 4.0). Oczywiście należy tutaj zwrócić uwagę, że proces rozszerzenia i modyfikacji języka był ciągły oraz uwzględniał charakterystyczne dla danego komputera różnice konstrukcyjne w sprzęcie i oprogramowaniu. Należy więc pamiętać, że jednak każdy z typów komputerów osobistych ma nieco inną wersję niby tego samego języka. W reklamie – firma Microsoft twierdzi, że 9 na 10 komputerów osobistych posiada ich wersję języka oraz istnieje 2 mln licencjonowanych instalacji. Ja im wierzę.

W 1981 roku powstała kolejna wersja języka GW (Gee Whiz) BASIC, wersja 5.0 przeznaczona dla komputerów osobistych z 16-bitowym mikroprocesorem. Wersja ta po porozumieniu firmy Microsoft z firmą IBM została pod nazwą BASICA zaimplementowana dla mikrokomputerów typu IBM PC. Język ten jest znany też jako BASIC-86 (również i tutaj 86 nie jest rokiem, a końcówką oznaczenia Intel 8086). Na tym jednak nie koniec tej historii. Twórcy języka, nieco oszołomieni tą karierą języka BASIC-80, postanowili walczyć o swoje prawa.

W 1974 roku pod auspicjami ANSI (American National Standards Institute) rozpoczął się nieco biurokratyczny proces opracowywania i zatwierdzania normy języka. Przewodniczącym Komitetu ANSI dla tych celów został profesor T. E. Kurtz. Niestety ten długotrwały proces standaryzacji jeszcze się nie zakończył. W skrócie – proponowana wersja standardu języka zawiera wszystkie jego podstawowe elementy z całkowicie inną niż w BASIC-80 metodą definiowania elementów grafiki opartą na standardzie GKS (Graphic Kernel System). W BASICu tym dołączono również bardzo efektywne i najnowocześniejsze w dziedzinie oprogramowania elementy współbieżności procesów umożliwiające stosowanie języka do oprogramowania systemów sterowania w czasie rzeczywistym.

Jednocześnie profesor T. E. Kurtz założył firmę True BASIC, która opracowała podzbiór języka BASIC-ANSI dla potrzeb komputerów osobistych i zaimplementowała go dla IBM PC.

I tu jeszcze nie koniec tej historii. Rozpoczął się wyścig różnych firm opracowujących coraz pełniejsze i efektywniejsze w użytkowaniu wersje języka uwzględniające najnowsze koncepcje w dziedzinie programowania. Wprowadza się deklarację różnego typu zmiennych o różnych strukturach danych (tablice, rekordy, wskaźniki), pełne postacie instrukcji strukturalnych, procedury i moduły.

Język BASIC zaczyna zatracać swój pierwotny ascetyczny charakter, zbliżając się do najnowocześniejszych języków programowania. Powstaje jednak pytanie, czy jest to jeszcze BASIC?

Zakres prezentacji języka BASIC. W tej książce omówiono część języka BASIC-80 (Microsoft BASIC wersja 4.0) okrojoną do postaci języka MBASIC wersja 2.0.

Problemy

2.1. Rozstrzygnij, którą z dwóch metod projektowania należy zastosować w realizacji podanych zadań i w których momentach należy zmienić metodę:

a) wykonanie linii metra,
b) układanie programu telewizyjnego,
c) projektowanie nowego typu samochodu,
d) nauka matematyki w szkole,
e) budowa nowego osiedla mieszkaniowego,
f) budowa nowoczesnego Centrum Sztuki,
g) układanie planu zajęć w szkole.

2.2. Podaj dla dostępnych tobie komputerów osobistych następujące parametry systemu:

a) konfiguracja systemu;
b) rodzaj ekranu – liczba wierszy i kolumn, liczba kolorów, sposób przyłączenia;
c) liczba klawiszy funkcyjnych w klawiaturze – odstępstwa od układu klawiatury maszyny do pisania;
d) umiejscowienie mikroprocesora i jego typ;
e) wielkość pamięci operacyjnej – podział na część stałą i zmienną;
f) sposób uruchomienia komputera;
g) inne dostępne języki programowania;
h) rodzaj translatora dla języka BASIC;
i) numer wersji języka BASIC.

Let's learn BASIC English

Dowiedzieliśmy się już, że językiem porozumiewania z komputerem jest język angielski, a dokładniej pewien dobrze określony jego podzbiór. Zanim więc nauczymy komputery języka polskiego, a realizacja tego postulatu nie jest w pełni możliwa, musimy się nauczyć języka komputerów. Język ten jest w pewnym sensie gwarą, znacznie upraszczającą wieloznaczność i ekspresyjność oraz bogactwo form zapisu języka potocznego.

Tym niemniej na wzór istniejącego Basic English (British American Scientific – International – Commercial = BASIC) – uproszczonego języka angielskiego do użytku międzynarodowego, powinniśmy znać uproszczony język informatyczny.

Na końcu kolejnych rozdziałów umieściłem więc słowniczki angielsko-polskie nowo wprowadzonych terminów, których znajomość ułatwi zrozumienie tekstów programów zapisanych w języku BASIC oraz czytanie oryginalnych podręczników dołączonych do każdego komputera. Ponadto zapobiegnie to używaniu żargonu informatycznego w stylu: „Weź skrecza i daj mu lejbel worka, do którego zdampuj trejs programu jako fajl alfa" – bez komentarza.

Zaczynamy więc studiowanie specjalistycznego słownika angielsko-polskiego.

basic podstawowy, fundamentalny

computer [wym. komputer, a nie kompjuter] komputer [*łac.*] polski rdzeń pochodzi od terminu komput czyli liczba, ilość, poczet; w Polsce XVII i XVIII w. początkowo ilość wojska uchwalona przez sejm; etat liczebny wojska stałego – źr. Encyklopedia Powszechna; *arch.* maszyna matematyczna, elektroniczna maszyna matematyczna; (ale na pewno nie należy używać terminu mózg elektroniczny, gdyż komputer jeszcze długo mózgiem nie będzie, a jak już się to stanie, to najpewniej będzie to mózg bioniczny)

screen, display ekran (monitora lub odbiornika telewizyjnego)

key klawisz

keyboard klawiatura z układem klawiszy

lineprinter drukarka

microprocessor mikroprocesor, jednostka przetwarzająca, procesor

bit bit; jednostka informacji przyjmująca dwie wartości – jest, nie ma (0, 1 lub **false** fałsz, **true** prawda)

byte bajt; oktet (złożony z uporządkowanych bitów reprezentujących wspólnie pewną, określoną kombinację ich wartości)

memory pamięć operacyjna

store pamięć, magazyn; **casette** ~ pamięć kasetowa

backup ~ pamięć pomocnicza; **mass** ~ pamięć masowa

secondary storage pamięć pomocnicza, drugorzędna

ROM (Read Only Memory) pamięć stała (umożliwia tylko odczytanie zapisanej w niej zawartości)

RAM (Random Access Memory) pamięć zmienna (o dowolnej sekwencji dostępów operacją zapisu lub odczytu zawartości)

EPROM (**Erasable Read Only Memory**) wymazywalna pamięć stała (możliwa jest wymiana jej zawartości w specjalnym urządzeniu)
hardware sprzęt komputera; dawniej żelastwo, sklep z częściami żelaznymi
software oprogramowanie (termin utworzony przez kontrast **hard** twardy, surowy, ostry, trudny; **soft** miękki, łagodny, przyjemny, delikatny, cichy; **ware** wyrób)
operating system system operacyjny, zarządzający
flowdiagram sieć działań
Top Down od ogółu do szczegółu
Bottom Up od szczegółu do ogółu
high level language język wysokiego poziomu
assembler asembler (spolszczony termin angielski)
translator translator; tłumacz (ten termin w informatyce się nie przyjął, chociaż **translation** tłumaczenie, translacja)
interpretator interpretator.
compiler kompilator (nieco zmieniony znaczeniowo termin określający osobę tworzącą coś nowego na zasadzie trochę tego, trochę tamtego i szczypta własnego talentu — tutaj jest to program)

END of lesson

3. Wprowadzanie tekstu programu

Po zapisaniu algorytmu w postaci programu możemy przystąpić do wprowadzenia tekstu programu do komputera, co może być wykonane tylko przy jego aktywnym współdziałaniu. Programem zarządzającym funkcjonowaniem komputera w tym okresie jest system BASIC.

Zanim omówimy zasady funkcjonowania systemu BASIC zacznijmy od prostej zabawy z komputerem. A więc włączamy (dokładnie według instrukcji) swój komputer osobisty. Kto chwilowo nim nie dysponuje, niech się skupi na tekście rozdziału.

3.1. Początek zabawy

Mam na początku poważny kłopot, gdyż nie chcę preferować żadnego typu komputera, dlatego też muszę czasem posługiwać się ogólnikami.

Komputery osobiste bez stacji dyskietek[1] mają umieszczony system BASIC w pamięci stałej. System ten zgłasza się natychmiast po włączeniu komputera wyświetlając na ekranie tekst typu:

```
Microsoft BASIC
©FIRMA, 1985
```

Komputery osobiste wyposażone w stacje dyskietek działają po włączeniu pod uniwersalnym systemem zarządzającym. System BASIC musi być wtedy sprowadzony na specjalne życzenie zleceniem na przykład typu:

```
A>BASIC
32765 Bytes Free
BASIC-80  Version 3.0
```

[1] Dyskietka jest nośnikiem informacji dla pamięci pomocniczej, podobna funkcjonalnie do kasety magnetycznej, ale o szybszym czasie dostępu i większej pojemności.

Po tych nagłówkach pojawia się zawsze pewien symbol zachęty – niestety również jego oznaczenia są różne: Ok, >, Ready,]. Tutaj wybierzemy pierwsze. Po symbolu zachęty pojawia się, często mrugający, symbol kursora wskazujący miejsce kolejno wprowadzanych przez nas znaków. A więc zaczynamy, nie przejmując się drobiazgami.

Dla rozróżnienia tekstu wyprowadzanego przez system BASIC na ekran od tekstu przez nas wprowadzanego, ten pierwszy będzie tutaj podkreślony (na ekranie oba się niczym nie różnią). Również dla uproszczenia zakończenie wprowadzonej linii będziemy na razie oznaczać symbolem ■, odpowiadającym wciśnięciu klawisza oznaczonego jako RETURN lub ENTER. Tekst po tym znaku, po prawej stronie jest moim komentarzem.

```
Ok
PRINT 5 ■                        'wyprowadz (wyswietl) wartosc 5,
 5                               'jest,
Ok
PRINT 3+5 ■                      'a teraz ile jest  3+5 ?
 8                               'zgadza sie !
Ok
PRINT 3.14 * 5.7 * 5.7 ■         'a teraz nieco trudniejsze wg. pi*r^2,
 102.0186                        'troche trudno sprawdzic
Ok                               ' - wierzymy na slowo,
LET  p=3.14 ■                    'stala pi nazwiemy  p i przypiszemy
Ok                               '  jej wartosc  3.14,
PRINT  p ■                       'sprawdzamy jaka jest wartosc  p ?
 3.14                            'zgadza sie !
Ok
PRINT  p * 5.7 * 5.7 ■           'obliczamy  pi*r^2 korzystajac z  p,
 102.0186                        'taki wynik otrzymalismy poprzednio,
Ok
LET  r=5.7 ■                     'druga wartosc przypiszemy do  r,
Ok
PRINT  p*r*r ■                   'i jeszcze raz sprawdzamy,
 102.0186                        'i znowu znajoma liczba,
Ok
LET  r=6.9                       'zmieniamy wartosc promienia,
Ok
PRINT  p*r*r ■                   'formule obliczania powierzchni okregu
 149.4954                        '  mamy juz w ogolnej postaci,
Ok
10  LET p=3.14 ■                 'teraz zmienimy sposob postepowania i
20  LET r=5.7  ■                 'nauczymy komputer co ma po kolei ro-
30  PRINT p * r * r ■            'bic; kolejne polecenia numerujemy, a
40  END ■                        'system polyka je bez odpowiedzi,
RUN ■                            'uruchamiamy program - RUSZAJ !
 102.0186                        'dostalismy znajomy wynik,
Ok
RUN ■                            'jeszcze raz !
 102.0186                        'i to samo, a wiec raz nauczony kompu-
Ok                               '  ter moze powtarzac czynnosci,
20 LET r=6.9 ■                   'poprawiamy instrukcje w linii 20,
LIST ■                           'sprawdzimy jak wyglada nasz program,
10 LET p=3.14
20 LET r=6.9
30 PRINT p * r * r
40 END
Ok                               'faktycznie Ok
RUN ■                            'a wiec RUSZAJ !
 149.4954
Ok                               'troche to uciazliwe zmienianie wartosci
20 INPUT r ■                     '  r, a wiec uzyjemy instrukcje WPROWADZ,
```

```
RUN ■                          'i RUSZAJ !
? 5.7                          'cos od nas chce - pewnie wartosc  r ?
 102.0186                      'zgadza sie,
Ok
RUN ■                          'i jeszcze raz ,
? 6.9                          'teraz juz wiemy,ze nalezy podac wartosc
 149.4954                      ' promienia  r ,
Ok
35 GOTO 20 ■                   'zapetlamy program - !?!
20 INPUT "Podaj promien "; r ■
30 PRINT "Powierzchnia= "; p$r$r ■
                                'poprawiamy tekst programu:
                               '35  zepetlenie programu !?!
                               '20  dodanie tekstu podpowiedzi
                               '30  objasnienie wyniku
                               'jak widac instrukcje mozna wprowadzac
                               'w dowolnej kolejnosci,
LIST ■
10 LET r=3.14
20 INPUT "Podaj promien "; r
30 PRINT "Powierzchnia= "; p$r$r
35 GOTO 20
40 END
Ok
RUN ■                           'sprawdzmy dzialanie programu,
Podaj promien ? 5.7 ■           'aha - teraz nam program podpowiada,
Powierzchnia=  102.0186 ■
Podaj promien ? 6.9 ■           'nastepny obieg programu,
Powierzchnia=  149.4954 ■
Podaj promien ?                 'wystarczy tego dobrego - warto
                                'przerwac dzialanie programu
                                'wciskamy klawisz CTRL i C
Break in 20                     'przerwanie w linii numer 20
Ok
NEW ■                           'wyczyszczenie obszaru pamieci
Ok
LIST ■                          'sprawdzmy
Ok                              'nie ma nic !
```

Teraz wprowadźmy do komputera tekst programu liczenia rolek tapety, podany w pierwszym rozdziale. Tekst wprowadzamy dokładnie (może być tylko dużymi literami). Jeżeli zrobiliśmy błąd w linii, wpisujemy ją jeszcze raz w dowolnym miejscu, byle z właściwym numerem. Po zapisaniu całego tekstu sprawdzamy go zleceniem LIST i ewentualnie poprawiamy. Dalej zakładamy, że mamy już program gotowy, a więc:

```
RUN■
? 2■                           'wg tekstu programu sa dwa pokoje,
? 2.75■                        'wysokosc mieszkania,
? 3.5, 5.5■                    'szerokosc * dlugosc  I  pokoju,
? 0.75, 2■                     'szerokosc * wysokosc  otworu,
Czy wszystkie otwory(TAK/NIE)? "NIE"■
? 3.5, 2.75■                   'szerokosc * wysokosc  otworu,
Czy wszystkie otwory(TAK/NIE)? "TAK"■
? 12, 6.5■                     'szerokosc * dlugosc  II pokoju,
? 0.75, 2■                     'szerokosc * wysokosc  otworu
Czy wszystkie otwory(TAK/NIE)? "TAK"■
liczba rolek tapety = 57       ' HURRA ! - mamy wynik
Ok
PRINT  pt■                     'jeszcze dowiemy sie o ogolnej
 152.4875                      '  powierzchni tapetowania,
Ok
PRINT  tp■
 2.75                          '  oraz powierzchni tapety.
Ok
```

Warto zapamiętać możliwość podglądnięcia wartości zmiennych programu, nawet po jego zakończeniu. Zagadnienia te będą szerzej wyjaśnione w dalszej części książki.

Na razie koniec zabawy. Wyłączamy komputery.

3.2. System BASIC

Rozumiemy już, że sam komputer w postaci zestawu sprzętu nie wystarcza do wprowadzania i wykonania programu. Konieczne jest wyposażenie go w pewien specjalny program umożliwiający jego działanie. Program ten nazwaliśmy systemem BASIC. Zaznaczam tutaj, że jest to pewne uproszczenie, ale dla naszych potrzeb wystarczające.

System BASIC jest programem z oprogramowania komputera służącym do sterowania pracą komputera w celu wprowadzenia, uruchomienia i wykonania programu zapisanego w języku BASIC. Program systemu jest zapisany na stałe w pamięci stałej lub musi być załadowany z pamięci pomocniczej.

Dalej założymy, że częścią naszego systemu BASIC znajdującego się w pamięci operacyjnej jest translator języka BASIC będący interpretatorem.

Po uruchomieniu systemu BASIC znajduje się on w stanie początkowym. Zawartość pozostałej pamięci operacyjnej jest pusta (dokładniej jest ona wypełniona wartościami równymi 0). System wyprowadzając znak zachęty znajduje się na *poziomie wprowadzania poleceń.*

Poleceniem dla systemu BASIC jest tekst wprowadzony w pojedynczej linii. Tekst polecenia jest zapisem czynności, której wykonania oczekujemy od systemu. Rozróżniamy dwa rodzaje poleceń:

1. *Zlecenie* jest poleceniem, które po wprowadzeniu ma być natychmiast wykonane przez system.
2. *Instrukcja* jest poleceniem, które po wprowadzeniu ma być zapisane w pamięci operacyjnej jako element programu.

W systemie BASIC rozróżnieniem rodzaju polecenia jest wystąpienie lub brak *numeru_linii* na początku wprowadzanej linii. Brak numeru oznacza zlecenie dla systemu. Większość czynności realizowanych w systemie może być zadawana jako zlecenia lub instrukcje. Wykonywanie tych czynności jako zleceń jest nazywane pracą w *trybie bezpośrednim* (*kalkulatorowym*).

Składnia linii

numer_linii słowo_kluczowe parametry koniec_linii

Numerem_linii jest liczba całkowita z zakresu 0...65535. Numery linii wyróżniają polecenia jako instrukcje programu i porządkują tekst programu odkładany w pamięci.

Słowo_kluczowe jest identyfikatorem czynności realizowanej jako zlecenie lub instrukcja. W języku BASIC czynności te są oznaczane charakterystycznymi słowami w języku angielskim (w książce słowa te będą zapisane dużymi literami). Postać *parametrów* polecenia jest określona składnią instrukcji języka.

Znakiem *końca_linii* jest znak sterujący RETURN lub ENTER. Wprowadzenie tego znaku kończy wprowadzanie linii i przekazanie jej jako polecenia do systemu BASIC.
Linia może zawierać od 0 do 255 znaków.

Po wprowadzeniu linii polecenia jest ono rozpoznawane przez system BASIC. Błędnie podane zlecenia są odrzucane i jest wyprowadzany komunikat o rodzaju błędu. Po wprowadzeniu polecenia będącego instrukcją nie jest wyprowadzony znak zachęty. Błędy w instrukcjach zostaną wykryte dopiero po uruchomieniu programu zleceniem RUN.

RUN
RUN *numer_linii*

Żądanie wykonania programu zapisanego aktualnie w pamięci operacyjnej. Brak parametru oznacza rozpoczęcie wykonywania od linii o najniższym numerze. Podanie *numeru_linii* wskazuje pierwszą instrukcję do wykonania.

Po uruchomieniu wykonywania programu system BASIC przechodzi na *poziom realizacji zlecenia*, przyjmując jedynie informacje żądane przez wykonywany program. Po zakończeniu wykonywania programu, w wyniku wykonania: ostatniej instrukcji, instrukcji końca programu lub wskutek wykrycia błędu, system BASIC wraca na poziom wprowadzania poleceń.

W trakcie realizacji zlecenia przez system BASIC możliwa jest ingerencja w jego działanie jedynie poprzez użycie klawiszy znaków sterujących.

Znaki sterujące interakcją z systemem:

(niektóre znaki mają różne oznaczenia na różnych klawiaturach)

CTRL-C[1], BREAK, ESC
– bezwarunkowe przerwanie wykonywania zlecenia.

CTRL-O – wstrzymanie wyprowadzania wyników wykonywania polecenia, bez zatrzymywania jego działania, aż do momentu kolejnego wprowadzenia CTRL-O.

CTRL-S – zawieszenie wykonywania zlecenia,

CTRL-O – wznowienie wykonywania zawieszonego zlecenia.

Działanie znaków sterujących dla zlecenia RUN oznacza odpowiednio przerwanie, wstrzymanie, zawieszenie lub wznowienie wykonywania programu.

RESET, CTRL-ALT-DEL[2]
– przerwanie aktualnych działań systemu i powrót do stanu początkowego ze zniszczeniem bieżącej zawartości zmiennej pamięci operacyjnej – zerowanie systemu.

[1] CTRL-C oznacza przy wciśniętym klawiszu sterowania oznaczonym jako CTRL (lub CONTROL) wciśnięcie klawisza litery C.
[2] CTRL-ALT-DEL oznacza równoczesne wciśnięcie wymienionych klawiszy.

Po przerwaniu wykonywania zlecenia znakiem CTRL-C system BASIC przechodzi na poziom wprowadzania kolejnego polecenia. Możliwość przerwania wykonywania programu ma znaczenie, jeżeli podejrzewamy, że program się zapętlił, czyli będzie działał bez końca. Również, niekiedy po dotychczas wyprowadzonych wynikach orientujemy się, że program został źle skonstruowany i w tej postaci jego dalsze wykonywanie jest bezsensowne. Chwilowe zawieszenie wykonywania programu za pomocą funkcji CTRL-S umożliwia spokojną analizę wyświetlonych na ekranie dotychczas otrzymanych wyników, zanim znikną one wyparte z ekranu przez następne. W niektórych systemach po wypełnieniu ekranu następuje automatyczne oczekiwanie na akceptację przez użytkownika, co jest sygnalizowane napisem *scroll.*

Całkowite przerwanie aktualnych działań systemu i jego zerowanie jest pewnego rodzaju radykalnym wyjściem z niejasnej sytuacji. Działanie to podejmujemy tylko wtedy, kiedy system zachowuje się niestandardowo lub chcemy rozpocząć całkiem nowe zadanie. Dla uniknięcia przypadkowego wyzerowania systemu (na przykład po kilku godzinach pracy), niektóre systemy zerują się dopiero przy równoczesnym wciśnięciu trzech różnych, odległych od siebie na klawiaturze, klawiszy.

Użytkowanie komputera osobistego. Korzystając z dobrego komputera, a tylko z takiego warto korzystać, nie należy się obawiać, że nasze pierwsze, trochę nieudolne próby zrobią mu jakąś krzywdę. Każda wprowadzona przez nas sekwencja znaków będzie przez komputer potraktowana poważnie i w przypadku wykrycia błędu zostanie on zasygnalizowany. Praktycznie każde nasze błędne postępowanie można poprawić, odtwarzając poprzednią postać. Czasem niestety jest to dość żmudna praca. Oprogramowania systemowego popsuć się nie da, gdyż znajduje się ono w pamięci stałej lub może być powtórnie załadowane z pamięci pomocniczej.

3.3. Redagowanie wprowadzonego tekstu

Postać wprowadzonej linii może być dowolna. Występujące w niej odstępy (spacje) nie mają wpływu na interpretację treści linii (z wyjątkiem spacji w napisach). Czasem jest wymagane, aby słowa kluczowe języka były otoczone co najmniej jedną spacją. Tekst linii może się zaczynać w dowolnym miejscu wiersza ekranu i może zajmować kilka wierszy przy wykorzystaniu znaku sterującego LINEFEED (tutaj oznaczonego jako ↓). Miejsce umieszczenia następnego znaku w linii jest oznaczone kursorem – kreską podkreślenia lub innym wyróżniającym się symbolem graficznym – często mrugającym.

Przykładowo:

```
100LETX=2 ■

100 LET  x = 2 ■

100 let  x  =   2 ■

1 0 0    L e T  X  =  2 ■

100 ↓
LET ↓
x=2 ■

100  LET ↓
          X = 2 ■
```

linie są sobie równoważne. Którą postać preferować, pozostawiam na razie do decyzji programującemu.

Dotychczas zakładaliśmy, że wprowadzając kolejne znaki tekstu linii nie popełniamy błędów. Jednak najbardziej zawodnym „elementem" komputera jest

człowiek. Szczególnie początkujący mają kłopoty w odnalezieniu właściwego klawisza lub też wciskają klawisze sąsiednie.

Najprostszym sposobem poprawienia błędu jest zakończenie dotychczas wprowadzonej linii znakiem RETURN i powtórne jej wprowadzenie. Jest to jednak sposób nieco kłopotliwy w zastosowaniu.

Korzystając z możliwości systemu BASIC, możemy przed zakończeniem linii znakiem RETURN dokonywać jej redakcji kasując ostatnio wprowadzone znaki, używając znaków sterujących.

Znaki sterujące i funkcyjne redagowanie linii:

(niektóre z nich mają różne oznaczenia na różnych klawiaturach)

SHIFT – wybór jednego z dwóch znaków związanych z danym klawiszem (górnego, jeżeli równocześnie jest wciśnięty klawisz SHIFT); dla klawiszy z literami jest wybierany w przypadku wciśnięcia SHIFT znak litery dużej.

CAPS_LOCK – klawisz stały blokowania wyboru znaków opisanych w górnej części klawiszy (w tym dużych liter) do czasu wyciśnięcia tegoż klawisza.

TAB, CTRL-I – przesunięcie kursora wprowadzania na początek kolejnej $n*8$ kolumny, $n = 0,1,2,...$

RUB, DEL – usunięcie ostatnio wprowadzonego znaku z wprowadzonej linii, bez skasowania go na ekranie.

DEL, CTRL-H – usunięcie ostatnio wprowadzonego znaku ze skasowaniem go na ekranie.

CTRL-R – przepisanie linii aktualnie wprowadzonej z pominięciem znaków usuniętych.

CTRL-U – usunięcie wszystkich znaków z linii aktualnie wprowadzonej.

LINEFEED – przesunięcie kursora wprowadzania w tej samej kolumnie o jeden wiersz w dół.

RETURN, ENTER – zaakceptowanie wprowadzonej linii i przekazanie jej pod kontrolę systemu BASIC.

CLEAR, HOME – zerowanie ekranu, ustawienie kursora w lewym górnym rogu ekranu.

W zestawie tych znaków sterujących i funkcyjnych spotyka się też w osobnej części klawiatury specjalne znaki sterujące edycji linii i fragmentu tekstu aktualnie wyświetlanego na ekranie

Rys. 3.1

– przesuwanie kursora wprowadzania odpowiednio w górę, w lewo, w dół i w prawo względem aktualnej pozycji; wyjście na dole lub na górze poza ekran powoduje odpowiednie przesunięcie fragmentu tekstu aktualnie wyświetlonego (rys. 3.1).

Ins – klawisz przełączający na tryb wprowadzania znaków począwszy od miejsca wskazanego kursorem, z odpowiednim rozsuwaniem tekstu; działanie tego klawisza jest wyłączane powtórnym jego użyciem lub użyciem jednego z klawiszy przesuwania kursora, wymazywania znaków Del lub akceptowania linii ENTER.

Del – klawisz wymazywania znaku wskazanego aktualnym położeniem kursora.

Korzystając z podanych znaków możemy szybko poprawić zauważone błędy we wprowadzanej linii. Różnica między znakiem funkcyjnym RUB a sterującym CTRL-H polega na tym, że w pierwszym przypadku usuwane znaki w kolejności odwrotnej są wpisywane między lewymi ukośnymi kreskami, a w drugim po prostu wymazywane z przesunięciem kursora w lewo. Kolejne bezpośrednio po sobie następujące wprowadzanie znaków RUB lub CTRL-H powoduje usuwanie znaków z linii w kolejności odwrotnej do ich wprowadzenia.

Rozważmy przykład pokazujący postać wyświetlanej linii na ekranie oraz kolejno wprowadzane znaki z klawiatury (rys. 3.2).

```
P          P_
R          PR_
A          PRA_
M          PRAM_
T          PRAMT_
RUB        PRAMT\T_
RUB        PRAMT\TM_
RUB        PRAMT\TMA_
I          PRAMT\TMA\I_
N          PRAMT\TMA\IN_
T          PRAMT\TMA\INT_
␣          PRAMT\TMA\INT␣_
Y          PRAMT\TMA\INT␣Y_
CTRL-H     PRAMT\TMA\INT␣_
X          PRAMT\TMA\INT␣X_
CTRL-X     PRINT␣X
```

Rys. 3.2

Tutaj podałem jedynie prosty przykład. Proponuję potrenować na prawdziwej klawiaturze z ekranem. Nabycie w tym wprawy przyda się potem przy wprowadzaniu tekstów programów. Proszę przy tym pamiętać, że tutaj szczególnie często widać odmienne koncepcje rozwiązań w różnych typach klawiatur. Zasady ogólne są zachowane, ale klawisze leżą w innych miejscach i mają inne oznaczenia oraz nie zawsze działają identycznie.

Oprócz redagowania pojedynczych linii istnieje też potrzeba redakcji tekstu, aktualnie istniejącego w pamięci programu.

Linie z instrukcjami programu możemy wprowadzać w dowolnej kolejności – w pamięci będą one zawsze porządkowane według wzrastającej wartości numeru linii. Warto więc założyć, że kolejne linie programu będą miały numery co

pewną wartość, np. co 10. Pozostawiamy sobie bowiem możliwość późniejszego wstawienia instrukcji dodatkowych, o których zapomnieliśmy w pierwotnej wersji. Dla usunięcia linii z tekstu programu wystarczy wprowadzić linię o numerze takim samym jak numer linii usuwanej. Inny sposób usuwania jest możliwy przez użycie zlecenia usuwania linii DELETE.

DELETE
DELETE *numer_linii*
DELETE *numer_linii_1* – *numer_linii_2*
DELETE – *numer_linii_2*

Żądanie usunięcia linii o podanym *numerze_linii* lub wielu linii, począwszy od linii o *numerze_linii_1* do linii o *numerze_linii_2* włącznie. Przy braku *numeru_linii_1* są usuwane linie począwszy od najniższego numeru. Zlecenie bezparametrowe oznacza usunięcie całego programu (*ostrożnie*!).

Po wykonaniu tego zlecenia, system BASIC wraca na poziom wprowadzania kolejnych poleceń.

Wielokrotne poprawianie tekstu programu w pamięci operacyjnej szybko doprowadza do sytuacji, że nie panujemy nad tym, jaka jest jego aktualna postać. Wykorzystujemy wtedy zlecenie LIST.

LIST
LIST *numer_linii*
LIST *numer_linii_1* – *numer_linii_2*
LIST *numer_linii_1*
LIST – *numer_linii_2*

Żądanie od systemu wyświetlenia treści wskazanych instrukcji zapisanych w pamięci operacyjnej. Parametry określają numery wyświetlanych linii. Pojedynczy *numer_linii* wskazuje daną linię. Wyświetlenie wielu linii jest określone przez podanie zakresu począwszy od *numeru_linii_1* do *numeru_linii_2*. Brak określenia zakresu oznacza numer linii o najwyższej lub najniższej wartości w programie. Zlecenie bezparametrowe oznacza wyświetlenie treści całego tekstu programu.

Zlecenie wyświetlania treści programu jest jednym z najczęściej używanych. Szczególnie przy braku drukarki, chęć sprawdzenia na bieżąco postaci określonych fragmentów programu zmusza do korzystania z tego zlecenia. Należy jednak zwrócić uwagę na efektywne posługiwanie się tym zleceniem, wykorzystując je w postaci z parametrami. Warto też pamiętać, że na ekranie mieści się określona

liczba ostatnio wyprowadzonych linii, a wszystkie wcześniejsze po prostu szybko przelecą przez ekran i znikną (w niektórych systemach, po zapełnieniu ekranu system oczekuje na zezwolenie przez użytkownika na kontynuowanie realizacji zlecenia).

Dla przykładu załóżmy istnienie w programie 100 linii numerowanych kolejno co 10 począwszy od 10, a więc 10, 20, 30,..., 1000 (taka numeracja linii jest dopuszczalna w języku BASIC).

Podane zlecenia spowodują odpowiednio:

LIST – wylistowanie treści wszystkich 100 linii,
LIST 50 – wylistowanie treści linii o numerze 50,
LIST 130–250 – wylistowanie treści 13 linii począwszy od linii o numerze 130, aż do linii o numerze 250,
LIST – 150 – wylistowanie pierwszych 15 linii,
LIST 910 – wylistowanie ostatnich 10 linii.

W trakcie realizacji zlecenia wyświetlania treści linii, możliwe jest skorzystanie ze znaków sterujących w celu:

CTRL-C – zaniechania dalszego wyświetlania,
CTRL-S – wstrzymania działania zlecenia dla dokładniejszego obejrzenia aktualnie wyświetlanego tekstu na ekranie, po czym wznowienie wykonywania zlecenia znakiem sterującym CTRL-Q.

Przed wprowadzaniem tekstu nowego programu poprzedni powinien zostać usunięty, w przeciwnym razie jego linie wymieszają się z nowymi.

NEW

Żądanie usunięcia całego tekstu programu z pamięci operacyjnej (*ostrożnie z tym zleceniem* – usuniętego tekstu nie można już potem odzyskać!).

3.4. Współpraca z pamięcią kasetową[1)]

Treść programu znajdująca się w pamięci operacyjnej po wyłączeniu komputera zostanie zniszczona. W najprostszych konfiguracjach komputerów do przechowywania treści programów służą kasety magnetofonowe. Z reguły są to typowe kasety o minimalnych długościach taśmy jak C-10 lub C-20.

Korzystając z pamięci kasetowej musimy pamiętać o następujących podstawowych zasadach jej funkcjonowania:

1. Zapis treści następuje począwszy od miejsca na taśmie aktualnie znajdującego się koło głowic.

[1)] Podane tutaj wiadomości są ogólnym opisem najczęściej spotykanych rozwiązań. Szczegółowe zasady są uzależnione od praktycznych rozwiązań sprzętowych.

2. Zapisanie nowej treści w środku taśmy może zniszczyć część zapisów znajdujących się za nią.
3. Współpraca komputera z magnetofonem musi się odbywać z udziałem użytkownika.

CSAVE *"nazwa_pliku"*

Wykonanie zlecenia powoduje zapisanie na taśmie kasetowej całej treści programu znajdującego się aktualnie w pamięci operacyjnej. Parametrem zlecenia jest *nazwa_pliku* (ujęta w cudzysłowy) zawierająca do kilku liter, która będzie poprzedzać zapis treści. Nazwa ta umożliwia wyszukanie odpowiedniego zapisu przy odczycie. Często tylko pierwsza litera nazwy jest brana pod uwagę przy identyfikacji pliku.

Przed zakończeniem wprowadzenia zlecenia należy przygotować magnetofon do zapisu (PLAY + RECORD lub START + ZAPIS). Po zakończeniu zapisu jest wyświetlany komunikat.

Składowanie treści programu na taśmie kasetowej jako pliku identyfikowanego *nazwą_pliku* jest pierwszym krokiem na drodze do wykorzystywania pamięci pomocniczych. Prostota tego rozwiązania niesie ze sobą konieczność zachowania porządku w bibliotece kaset zawierających na każdej z nich od jednego do wielu programów. Należy też pamiętać o odpowiednim przechowywaniu kaset, z dala od zgubnego wpływu urządzeń elektrycznych, elektronicznych, cieplnych itp.

CLOAD *"nazwa_pliku"*

Wykonanie zlecenia powoduje wyszukanie na taśmie kasetowej początku pliku o podanej *nazwie_pliku* i następnie przepisanie treści programu z taśmy do pamięci operacyjnej. Dotychczas istniejąca w pamięci treść programu zostaje zniszczona. Brak *nazwy_pliku* oznacza załadowanie pierwszego spotkanego na taśmie programu. Często tylko pierwsza litera nazwy jest brana pod uwagę przy identyfikacji pliku.

Po wprowadzeniu zlecenia należy przygotować magnetofon do odczytu (PLAY lub ODCZYT). W trakcie poszukiwania pliku z programem oraz podczas jego wprowadzania na ekranie są wyświetlane komunikaty o stanie wykonywania tego zlecenia.

Obie operacje – składowanie i *ładowanie* programu dla pamięci kasetowej – są dość czasochłonne, prócz tego często mogą wystąpić błędy zapisu lub odczytu. Nie jest to więc pamięć pewna i niezawodna. Stąd też, pomimo dość

dużego czasu składowania warto jest ten sam, *niezwykle ważny dla nas program*, zapisać na tej samej lub kilku kasetach wielokrotnie. Następnie warto też zapis sprawdzić z oryginałem wykorzystując odpowiednie polecenia (np. CLOAD? "*nazwa_pliku*" lub VERIFY "*nazwa_pliku*"). Podane zasady współpracy z pamięcią kasetową zilustrujemy przykładem składowania na kasecie programu obliczającego liczbę rolek tapet.

```
LIST
.....                          'tutaj zostal wylistowany caly
.....                          ' tekst programu dla kontroli,
Ok
CSAVE  "tapeta"                'skladowanie, a wiec nalezy
Press PLAY and REC and next any key:
                               'wcisnac PLAY+REC na magnetofonie !
                               ' i dowolny klawisz,
                               'TERAZ CIERPLIWIE CZEKAMY !
Ok                             'udalo sie !
                               'sprawdzimy jakosc zapisu:
                               '- przewijamy tasme przed poczatek
                               '  zapisu  <<- REW.
CLOAD?  "tapeta""              'sprawdzanie , a wiec nalezy
Press PLAY and next any key:
                               'wcisnac PLAY
                               ' i dowolny klawisz
Found: BUFET                   'czytany jest naglowek wczesniejsze-
                               ' go pliku o nazwie  BUFET
Found: tapeta                  'czytany jest nasz plik,
Ok                             'w porzadku - jesli pojawilby sie
                               ' komunikat o bledzie to nalezy rozpo-
                               ' czac od skladowania,
NEW                            'usuwam program z pamieci,
Ok
                               'przewijamy tasme na poczatek lub
                               ' w miejsce poprzedzajace poszukiwa-
                               ' ny plik, uzywajac  REW  lub  FOR.
CLOAD "tapeta"                 'ladowanie programu,  wiec
Press PLAY and next any key:
                               'wciskamy PLAY i dowolny klawisz.
                               'TERAZ CZEKAMY CIERPLIWIE
                               ' sluchajac buczenia magnetofonu !
Ok                             'udalo sie,
LIST                           'sprawdzmy wynik ladowania
  ???                          'czy to samo ?
Ok
```

Teraz możemy już sami ładować z wypożyczonych od znajomych kaset programy gier komputerowych oraz inne bardzo użyteczne i ciekawe programy. Jeżeli zaś uzyskaliśmy wyraźną zgodę pożyczającego, to możemy programy te, mając tylko jeden magnetofon, zapisać na własne kasety.

Kopiowanie cudzych programów jest dopuszczalne jedynie po uzyskaniu licencji lub wyraźnej zgody pierwotnego właściciela – najczęściej autora programu. Każde inne zachowanie jest *naruszeniem norm prawnych i etycznych.*

Styl programowania

Styl programowania jest pojęciem abstrakcyjnym, określającym zespół warunków, które determinują otrzymanie programu nie tylko poprawnego, ale również i ładnego oraz eleganckiego pod względem zasad metodyki programowania. Rozważając te zagadnienia będziemy zajmować się zawsze programem poprawnym, czyli takim który:

- dla dobrze określonych zbiorów danych dokonuje ich przetworzenia w zbiór wyników zgodnie z założonym algorytmem, opracowanym na podstawie specyfikacji problemu przewidzianego do rozwiązania z wykorzystaniem komputera,
- dla wszystkich innych zestawów danych, niezgodnych z założeniami, spowoduje wyprowadzenie komunikatu o braku możliwości uzyskania wyników.

Taka definicja programu poprawnego, jak na nasze potrzeby, jest na razie wystarczająca. Pominiemy tutaj problemy weryfikacji programów, czyli metod stwierdzenia i udowodnienia ich poprawności.

Obecnie program możemy przyrównać do dzieła artysty, np. obrazu. Oczywiście przesłanki stwierdzające, że ten obraz nam się bardzo podoba, a ten mniej, są oparte na subiektywnych gustach oceniającego. Podobnie jest z wyrażeniem opinii porównującej elegancję dwóch lub więcej programów realizujących ten sam algorytm. Przyjmiemy przy tym, że ich techniczne własności – takie jak czas wykonania i zajętość pamięci – są porównywalne.

Przy omawianiu stylu programowania będziemy się więc zajmować postacią tekstu programu, tutaj zapisanego w języku BASIC. Na podstawie wypracowanych już w informatyce podstawowych zasad konstruowania ładnych, czytelnych i zrozumiałych programów będziemy analizować, które z zastosowanych elementów języka, i w jaki sposób, mają wpływ na stworzenie programu ocenianego jako elegancki. Na szczęście, określenie elegancja nie jest tutaj wynikiem szybko zmieniającej się mody, a logicznym następstwem doświadczeń wielu programistów. Pozostaje jeszcze wskazać motywację skłaniającą programistę do pisania programów eleganckich.

W tym przypadku, odwołując się również do doświadczeń, należy stwierdzić, że w trudnym procesie konstruowania, uruchomienia i weryfikacji oraz ewentualnych modyfikacji programu posługiwanie się tekstem programu eleganckiego ułatwia i przyśpiesza ten proces. Na każdym z tych etapów czytelność tekstu oraz jego zrozumiałość w sensie przeznaczenia i funkcji każdego elementu programu chroni przed popełnieniem wielu błędów, a także dostarcza większej liczby informacji pozwalających na zlokalizowanie i usunięcie istniejących błędów.

Czytelników tych rozdziałów zachęcam więc do uważnej analizy umieszczanych tam kanonów dobrego stylu programowania. Być może na początku niektóre z nich będą się wydawały zbędne lub wręcz utrudniające szybkie napisanie programu. Proszę jednak uwierzyć, na razie na słowo, że poniesiony od samego początku trud opłaca się i co więcej wiedza ta w przyszłości w dużym stopniu może być wykorzystana w programowaniu w innych językach.

Bez względu jednak na stopień akceptacji tych stwierdzeń nie wolno zakładać metody programowania typu „Quick and Dirty", czyli szybko i paskudnie, nawet w stosunku do programów, które mają znaleźć się szybko w koszu. Okazuje się bowiem, że często takie programy są potem długo używane i co więcej modyfikowane i rozbudowywane, zachowując w swej strukturze błędy dzieciństwa.

Sytuacje błędne

Stwierdzenia niektórych programujących, że udało im się natychmiast po wprowadzeniu uruchomić program i był on bezbłędny należy traktować z przymrużeniem oka. W praktyce, z wyjątkiem niezwykle rzadkich przypadków graniczących ze szczęściem, sytuacja taka nie występuje. Zanim więc otrzymamy program poprawny, musimy go wielokrotnie poprawić usuwając z niego błędnie zapisane konstrukcje języka, a także pomyłki w algorytmie.

Dla uzyskania jednoznaczności w zrozumieniu przez komputer sekwencji czynności, jakie ma on wykonać rozwiązując zadany przez nas problem, w języku programowania są określone ścisłe zasady konstruowania poleceń. Reguły składni języka, czyli jego ortografia i gramatyka, nie dopuszczają żadnych odstępstw. Jednakże człowiek jest omylny i często popełniając błąd nie od razu zdaje sobie z tego sprawę. Na szczęście system działający w komputerze wspomaga nas wskazując nam miejsca w programie, które są niezgodne z przyjętymi zasadami konstrukcji zleceń i instrukcji języka.

Wiedząc o tym wszystkim, musimy już na etapie przygotowywania algorytmu, a następnie uruchamiania programu przygotować się na to, że oprócz poprawnych konstrukcji nieświadomie wprowadzamy konstrukcje błędne. Dokładniejsze omówienie naszego postępowania w przygotowaniu programu do szybkiego sprawdzenia jego poprawności i usunięcia błędów zostanie podane w rozdziale 10.

Tutaj i w następnych punktach pod tym tytułem będziemy komentować przyczyny błędów związanych z nieprawidłowym zapisem konstrukcji języka. Rozróżniamy *statycznie* tkwiące w tekście programu *błędy składni* (ortograficzne i gramatyczne) oraz *błędy wykonywania* programu wykrywane w trakcie *dynamicznego* działania programu dla zadanego zbioru danych.

Wykrycie i zasygnalizowanie obu rodzajów błędów jest rolą translatora języka. W przypadku wykorzystywania w systemie BASIC interpretatora języka

oba rodzaje błędów są wykrywane w trakcie wykonywania, chociaż inna jest ich przyczyna. Wykrycie błędu w programie jest sygnalizowane odpowiednim komunikatem na ekranie i z reguły jest przerywane dalsze jego wykonywanie. Komunikaty błędów, które mogą być wykryte przez system BASIC są podane w dodatku B.

Tutaj zajmiemy się błędami, które mogą powstać dla instrukcji i zleceń dotychczas omówionych.

Syntax error

Błąd w składni linii o podanym numerze spowodowany wprowadzeniem złego znaku, zagubieniem innego lub ich przestawieniem. Tekst linii zawiera więc błędy typu ortograficznego uniemożliwiające interpretację polecenia. Linię tę należy powtórnie wpisać, wprowadzając najpierw znak Q – przenosząc tym samym system BASIC na poziom wprowadzania poleceń. Przykładami takich błędów są:

[1] DALETE 30 – nierozpoznawalne słowo kluczowe,
LIST A20 – parametr może być tylko liczbą całkowitą,
RUN NEW – dwa polecenia w jednej linii.

Undefined line

Niezdefiniowana linia, czyli w poleceniu DELETE występuje jako parametr numer linii nie istniejącej w tekście programu.

Komunikaty o błędzie w tekście programu są z reguły uzupełnione *numerem_linii*, w której błąd ten został wykryty.

Problemy

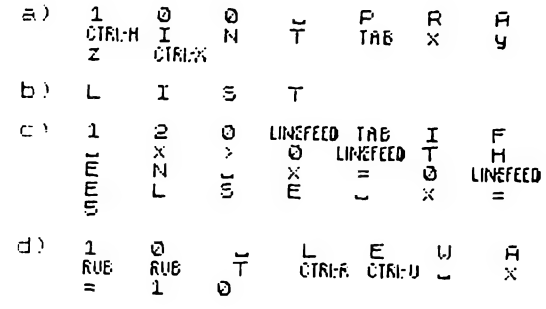
3.1. Podaj co zostanie wylistowane, jeżeli wprowadzono kolejno następujące linie:

```
a) NEW
   LIST
   30  INSTR_A
   40  INSTR_B
   10  INSTR_C
   LIST
   15  INSTR_D
    5  INSTR_E
   DELETE  40
   33  INSTR_F
   LIST  10-30
   10  INSTR_F
   12  INSTR_G
   LIST
```

```
b) NEW
    100  INSTR_a
     90  INSTR_b
     80  INSTR_c
     70  INSTR_d
     60  INSTR_e
     50  INSTR_f
   LIST
     95  INSTR_x
   DELETE  -70
     55  INSTR_y
    110  INSTR_z
   INSTR_q
   LIST
```

[1] Patrz uwaga: str. 9.

3.2. Podaj postacie linii na ekranie po wprowadzeniu kolejno znaków (rys. 3.3)



Rys. 3.3

3.3. Zaprojektuj zestaw poleceń mających na celu poprawienie w programie zapisanym na kasecie pod nazwą "ABECE", 25 i 75 linii wpisując instrukcję LET Z = 0, a następnie składowanie nowej wersji programu pod nazwą "XYZ"

Let's learn BASIC English

instruction polecenie![1]; instrukcja
command zlecenie!; komenda; rozkaz
statement instrukcja!; zdanie
order rozkaz!; zlecenie
prompt zachęta; znak zachęty
line number numer linii
keyword słowo kluczowe
eol (end of line) koniec linii (logiczny koniec linii)
RUN tutaj RUSZAJ, URUCHOM; (mniej pasuje biec, jechać, kursować)
DELETE USUŃ
LIST LISTUJ; wyświetl; (rzadziej pokaż)
NEW ODNÓW (raczej trzeba utworzyć tryb rozkazujący); nowy
CSAVE (Casette SAVE) SKŁADUJ na Kasecie; zachowaj
CLOAD (Casette LOAD) ŁADUJ z Kasety
syntax składnia
execution wykonanie
runtime wykonywanie
error błąd; **syntax** ~ błąd składniowy; **runtime** ~ błąd wykonywania
undefined niezdefiniowana

[1] Podane tłumaczenia terminów oznaczone ! są pewną propozycją w dotychczas panującym bałaganie; bardzo często spotykamy tłumaczenia przypisujące drugie lub trzecie znaczenia; również w oryginalnych podręcznikach angielskich różne firmy i różni autorzy przypisują różne znaczenia tym samym terminom (nie jest to jedyny taki przypadek w terminologii informatycznej).

mark, sign, character znak (na ten termin nie ma rady — tutaj język polski jest za ubogi; jeżeli będzie to konieczne możemy stosować przymiotniki) **mark** znak z oznaczenia (ptaszek); **sign** znak liczby;

character znak tekstu; **special** ~ znak specjalny;
function ~ znak funkcyjny; **control** ~ znak sterujący

Copyright Prawo kopiowania (tradycyjne oznaczenie zastrzeżenia kopiowania danej rzeczy bez zgody jej autora lub właściciela)

Również oznaczenia niektórych klawiszy na klawiaturze mają swój źródłosłów w terminach angielskich.

Znaki sterujące:

RETURN (carriage return) WPROWADŹ, zakończenie wprowadzania linii; powrót karetki (przesunięcie wałka w maszynie do pisania — tutaj uzyskało nowe znaczenie)
ENTER WPROWADŹ (równoważne RETURN)
ESCAPE WYZWÓL
RUBout WYKREŚL ostatnio wprowadzony znak
DELete WYMAŻ znak
LINEFEED ZMIEŃ wiersz
TABulate TABulacja (znana z maszyn do pisania)
BREAK PRZERWIJ
SHIFT ZMIENIACZ (do wprowadzenia pojedynczego znaku z drugiego zestawu znaków)
CAPS LOCK (CAPitalize) BLOKUJ DUŻE (czyli blokuj na wprowadzanie drugiego zestawu znaków)
CTRL (ConTRoL) STERUJ

Znaki sterujące CTRL→ (niektóre)

C (Cancel) Skasuj
O (Output) Wstrzymaj wyprowadzanie
R (Retype) Przepisz
S (Suspend) Zawieś
U (Update) Uaktualnij, Usuń

END of lesson

4. Język Mikro BASIC

Zaczynamy od początku. Po wprowadzeniu podstawowych pojęć określających otoczenie, w jakim będziemy konstruować i wykonywać programy, zajmiemy się prezentacją krok po kroku kolejnych elementów języka BASIC.

4.1. Alfabet języka

Zbiór znaków tworzących alfabet języka i systemu BASIC dzieli się na cztery grupy.

Zbiór znaków języka:

1. Duże i małe *litery* alfabetu angielskiego:
   A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
   a b c d e f g h i j k l m n o p q r s t u v w x y z
2. *Cyfry:*
   0 1 2 3 4 5 6 7 8 9
3. *Symbole*[1]:
   ␣ = + − * / ↑ ( ) % # $!
   [ ] , . ′ " : & ? < > \ @ _
4. *Znaki sterujące i funkcyjne*

W wielu komputerach są wykorzystywane jedynie duże litery. Często też co prawda możemy tekst programu pisać dużymi i małymi literami, ale system dla swojej wygody przekształca zapis tekstu na duże litery.

[1] ␣ oznacza odstęp (spację); oznaczenie to jest używane w tych miejscach, gdy konieczne jest jawne zaprezentowanie spacji.

W alfabecie brakuje też liter występujących w naszym języku, takich jak ą, ę, ć, ł, ń, ó, ś, ź, ż (brakuje również liter używanych w innych językach, takich jak ä, ü, š itp.). Na razie musimy się z tym pogodzić, chociaż w przyszłości krajowe komputery będą miały pełen alfabet (przynajmniej mam taką nadzieję).

W zapisie tekstów programów w książce będę korzystał z dużych i małych liter dla wyróżnienia elementów języka oraz, gdzie tylko mogłyby to być dopuszczalne, tekst będę zapisywać w języku polskim.

W języku programowania symbole są używane do oznaczania:
a) *operatorów* operacji arytmetycznych i relacji,
b) *separatorów* rozdzielających elementy składni języka,
c) *atrybutów* elementów języka.

Do oznaczania operatorów, separatorów i atrybutów są też używane znaki będące złożeniem dwóch symboli lub tworzące słowo z liter. Wiele spośród symboli używanych dotychczas w matematyce oraz w interpunkcji zachowało swoje pierwotne znaczenie. Pojawiły się też nowe znaki lub też stare nabyły nowych znaczeń.

Operatory:

=	równość, przypisanie	↑	potęgowanie
+	plus, konkatenacja	(	lewy nawias
−	minus	)	prawy nawias
*	mnożenie	<	mniejszość
/	dzielenie	>	większość

Separatory:

␣	odstęp, spacja	"	cudzysłów
,	przecinek	.	kropka dziesiętna
;	średnik	#	krzyżyk

Atrybuty:

$	symbol dolara (waluty)	?	pytajnik

Powyżej wymieniono tylko symbole wykorzystywane w tym opisie języka.

Znaki sterujące zostały już zaprezentowane w poprzednim rozdziale przy omawianiu znaczenia klawiszy funkcji specjalnych.

Korzystając z alfabetu języka konstruujemy podstawowe elementy języka używane następnie w tekście linii. Znamy już dwa elementy składni języka: *numer_linii* oraz *identyfikator słowa_kluczowego.*

Elementy języka BASIC. W parametrach poleceń (instrukcji lub zleceń) korzystamy z następujących elementów języka BASIC:
- stałych,
- zmiennych,
- operatorów,
- separatorów,
- parametrów funkcjonalnych.

4.2. Typy i stałe

Wykonywanie programu przez komputer polega na przetwarzaniu danych będących pewnymi wartościami.

Typy wartości. Na razie w języku BASIC wyróżniamy dwa typy wartości:
1. *Typ liczbowy* obejmuje wszystkie wartości liczb rzeczywistych, dodatnich i ujemnych z przedziału ograniczonego możliwościami komputera.
2. *Typ tekstowy* obejmuje wartości będące napisami złożonymi z ograniczonej implementacyjnie sekwencji dowolnych znaków.

W matematyce *liczbami rzeczywistymi* są liczby wymierne (liczby całkowite i ułamkowe) oraz *liczby niewymierne*. Ze względu na właściwości funkcjonowania komputera możliwe jest zapisanie w pamięci oraz przetwarzanie liczb jedynie o ograniczonym stopniu precyzji. W zapisie liczb *cyframi znaczącymi* są wszystkie cyfry, nie będące zerami i cyfra zero znajdująca się między cyframi znaczącymi lub na końcu liczby całkowitej.

Zakres wartości. W języku BASIC wartości typu liczbowego są akceptowane z precyzją do 7 cyfr znaczących, a zapamiętywane i przetwarzane z precyzją do 6 cyfr znaczących. Zakres tych wartości musi zawierać się w przedziale

$$(-1 \cdot 10^{38} \ldots -1 \cdot 10^{-38},\ 0,\ 1 \cdot 10^{-38} \ldots 1 \cdot 10^{38})$$

gdzie wartości większe od $-1 \cdot 10^{-38}$, a mniejsze od $1 \cdot 10^{-38}$ są utożsamiane z wartością 0.

Podany zakres wartości jest wynikiem zapisu wartości w postaci *liczb zmiennoprzecinkowych* złożonych z mantysy zapisywanej z precyzją do 6 cyfr znaczących o wartości należącej do przedziału [0, 1] i wykładnika dziesiętnego należącego do przedziału [−38, +38].

Stała reprezentuje wartość określonego typu liczbowego lub tekstowego zapisywaną bezpośrednio jako element języka.

Stałe typu liczbowego mogą być zapisane w jednej z trzech postaci.

1. *Postać całkowita liczby* jest ciągiem cyfr, ewentualnie poprzedzonym znakiem plus lub minus. Poprawne są zapisy:

0	10	1234567	00321
+0	1273	−999999	+1

2. *Postać ułamkowa liczby* jest złożona z dwóch ciągów cyfr (jeden z nich może być pusty) rozdzielonych kropką i ewentualnie poprzedzonych znakiem plus lub minus. Poprawne są zapisy:

12.37	10.	−0.0	.00027
−1.237	.01	+.1237	.000099

Znak kropki oddzielający część całkowitą od dziesiętnej jest zwyczajowo przyjęty w krajach anglosaskich — musimy więc i tutaj wyzbyć się szybko naszych przyzwyczajeń ze szkoły.

3. *Postać wykładnicza liczby* jest postacią całkowitą lub ułamkową uzupełnioną o wykładnik dziesiętny oznaczony symbolem litery E (lub e) i wartością całkowitą dodatnią lub ujemną. *Wykładnik dziesiętny* podaje wartość potęgi przy podstawie równej 10 przez jaką należy pomnożyć wartość ją poprzedzającą (mantysę), aby otrzymać rzeczywistą wartość stałej. Poprawne są zapisy:

0E+0	1E+5	1234567E−20
12.37E−1	−1E−5	1234567E+20

Przykładowo więc zapis 12.37E−1 jest zapisem $12.37_{10}-1$ czyli $12.37 \cdot 10^{-1}$, co jest równe 1.237. (Ta druga kropka kończy zdanie).
Postać wykładnicza liczby jest użyteczna do zapisu bardzo małych lub bardzo dużych wartości.

W tekście programu wartości stałych mogą być zapisywane w najdogodniejszej postaci. Równoważne co do wartości są postacie zapisu liczby:

0	0.0	0E+0
1237	1237.	12.37E+2
	−163.45	−0.16345E+3
	−0.000457	−457E−6

Stałe typu liczbowego. Reguły zapisu.

1. Użycie znaku plus jest nieobowiązkowe.
2. System akceptuje jedynie 7 cyfr znaczących, pozostałe są pomijane.
3. Część całkowitą od części ułamkowej rozdziela symbol kropki.
4. Zerowa część całkowita liczby ułamkowej może być pominięta.
5. Wykładnik musi być poprzedzony symbolem litery E.
6. Wykładnik dodatni może być pozbawiony znaku plus.

Stałe typu tekstowego są zapisywane jako *napisy* w postaci sekwencji znaków ujętych w cudzysłowy. Napis może zawierać od 0 do 255 znaków. W niektórych przypadkach – jeżeli w napisie nie występują znaki separatorów oraz znaki sterujące – cudzysłowy mogą być pominięte. Poprawnymi napisami są:

```
"BASIC"      "-123.75"      "# *3$"
"Alfabet"    "a+b*c"        "a, b, c, d, e"
```

Należy tutaj zwrócić uwagę na to, że zapis stałej typu liczbowego jest zapisem reprezentacji wartości liczbowej, a zapis stałej typu tekstowego jest zapisem tylko sekwencji znaków. Stąd, dla przykładu, gdy

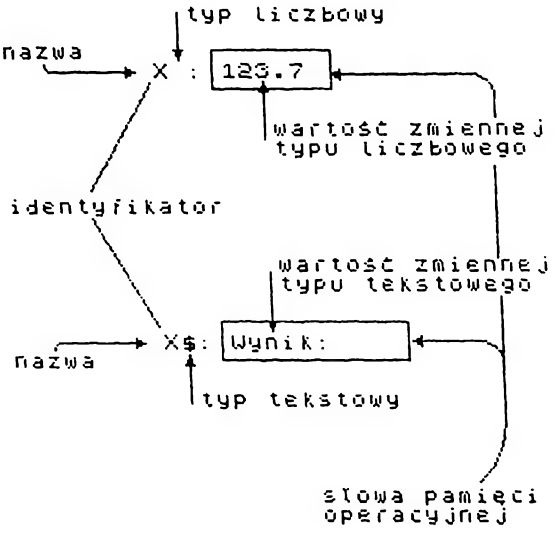
```
-123.75     jest stałą typu liczbowego, a
"-123.75"   jest stałą typu tekstowego
```

to obie te stałe są zapisywane w pamięci komputera w całkiem innych postaciach i nie jest na przykład możliwe bezpośrednie ich zsumowanie.

4.3. Zmienne

W trakcie wykonywania programu wartości są przechowywane w pamięci operacyjnej. W języku programowania miejsca ich przebywania są traktowane jako zmienne identyfikowane nazwami. Pojęcie zmiennej obejmuje więc kilka elementów (rys. 4.1):

1. Nazwa zmiennej identyfikuje miejsce położenia zmiennej w pamięci operacyjnej.
2. Wartość zmiennej jest zawartością słów pamięci operacyjnej przypisanych danej zmiennej.
3. Typ zmiennej określa typ wartości zmiennej, czyli sposób interpretacji zapisu wartości.
4. Atrybuty zmiennej specyfikują dodatkowe własności zmiennej z punktu widzenia ograniczeń języka.



Rys. 4.1

Identyfikator zmiennej. Podstawą symbolicznej nazwy zmiennej jest identyfikator konstruowany według następujących zasad:

1. Pierwszym znakiem musi być zawsze litera.
2. Drugim znakiem może być cyfra lub litera.

W niektórych bardzo uproszczonych wersjach języka BASIC drugim znakiem może być co najwyżej cyfra. Dostępne jest więc wtedy tylko 286 różnych identyfikatorów. Poprawnymi nazwami zmiennych są:

a	*z*	*x*	*v*	*c*	*i*	*p*
b0	*b9*	*xy*	*v5*	*cd*	*i2*	*pi*

W identyfikatorze mogą występować małe i duże litery, znaczeniowo ze sobą utożsamiane. My będziemy dalej na nazwy zmiennych używać małych liter.

Analogicznie do typów stałych wyróżniamy na razie dwa typy wartości zmiennych prostych.

Zmienne typu liczbowego służą do przechowywania wartości binarnych interpretowanych jako wartości liczbowe.

Zmienne typu tekstowego przechowują sekwencje znaków tworzących napisy.

Dla rozróżnienia typu zmiennej, identyfikator zmiennej typu tekstowego musi być uzupełniony symbolem $.

Ten sam identyfikator zmiennej może być wykorzystany dla utworzenia nazw zmiennych różnego typu. Poprawne są więc nazwy zmiennych typu:

– liczbowego: *a b z1 a0 c9 cd*
– tekstowego: *a$ b$ x$ a1$ c9$ cd$*

Pierwsze wystąpienie nazwy zmiennej w programie jest równoważne jej zadeklarowaniu. *Deklaracja* (tutaj niejawna) zmiennej jest informacją dla systemu BASIC o konieczności przydzielenia w pamięci określonej liczby słów na przechowywanie wartości oraz powoduje przypisanie nazwie zmiennej adresu tych słów. Początkowo po uruchomieniu programu wszystkie zmienne typu liczbowego mają wartość równą 0, a typu tekstowego wartość równą *napisowi pustemu* " ".

4.4. Instrukcja przypisania

Wprowadzenie nowej wartości do zmiennej następuje po wykonaniu *instrukcji przypisania.*

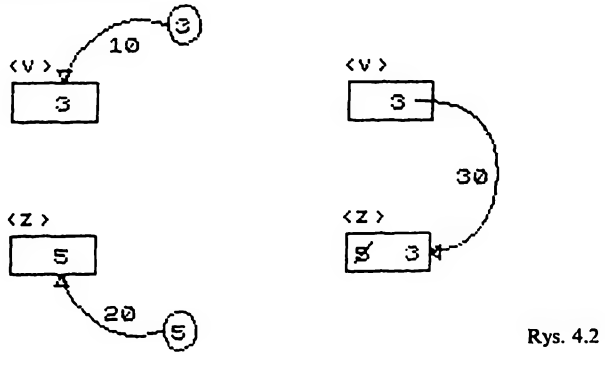
LET *zmienna = wartość*

Zmiennej o nazwie podanej po lewej stronie znaku równości zostanie przypisana *wartość* z prawej strony – wyrażona bezpośrednio stałą lub pobrana ze zmiennej o nazwie tam podanej. Po wykonaniu instrukcji, poprzednia wartość zmiennej z lewej strony zostaje zniszczona. Wymagane jest, aby typ zmiennej i wartości był identyczny.

Wykonanie instrukcji

```
10  LET  v=3
20  LET  z=5
30  LET  z=v
```

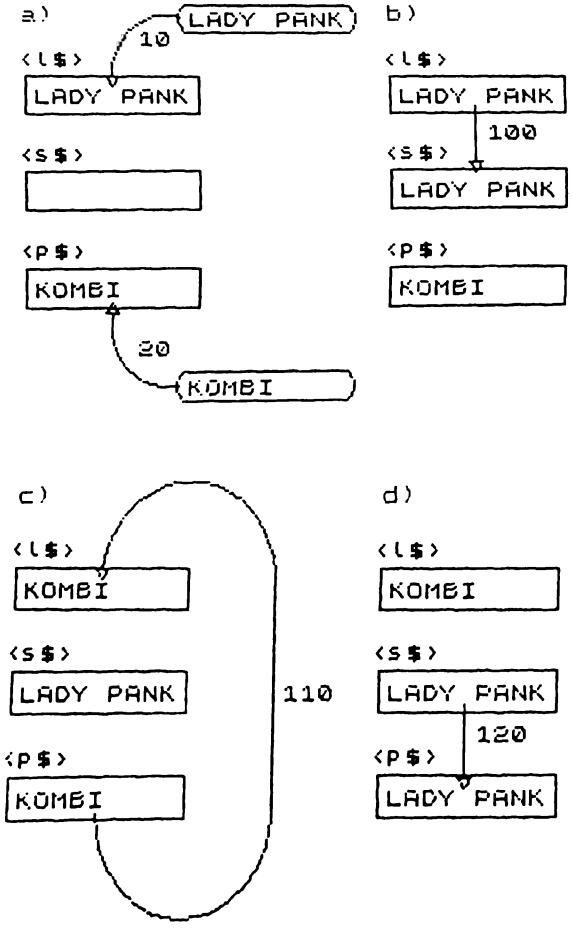
spowoduje (rys. 4.2):



Rys. 4.2

10 – przypisanie zmiennej o nazwie *v* wartości 3,
20 – przypisanie zmiennej o nazwie *z* wartości 5,
30 – przypisanie zmiennej o nazwie *z* wartości zmiennej o nazwie *v*, czyli wartości 3 z zamazaniem starej wartości równej poprzednio 5.

Rozważmy teraz następującą sytuację. Trzymając lewą ręką radio, aby nie spadło, chcemy zamienić miejscami kasety w dwóch magnetofonach – lewym (*l*) i prawym (*p*). Możemy to wykonać tylko w ten sposób, że wyjmujemy kasetę z lewego magnetofonu i kładziemy ją na stole (*l*→ *s*). Następnie wyjmujemy kasetę z prawego magnetofonu i wkładamy ją do lewego (*p*→*l*), i wtedy bierzemy kasetę ze stołu i wkładamy ją do prawego magnetofonu (*s*→*p*) (rys. 4.3).



Rys. 4.3

W języku BASIC czynności te zapisujemy następująco:

```
5   '                                stan poczatkowy
10  LET  l$="LADY PANK"
20  LET  p$="KOMBI"
50  '...
100 LET  s$=l$              ' LADY PANK  odkladamy na stol,
110 LET  l$=p$              ' KOMBI    z prawego do lewego,
120 LET  p$=s$              ' LADY PANK          do prawego.
130 '...
```

4.5. Instrukcje wprowadzania danych

W języku BASIC dane dla programu możemy zgrupować w zestawie danych opisanych *instrukcjami danych.*

DATA *lista_wartości*

Na *liście_wartości* jest umieszczona dowolna liczba wartości typu liczbowego lub tekstowego oddzielonych od siebie przecinkami. Wszystkie instrukcje danych uporządkowane według wzrastających numerów linii tworzą zestaw danych zwany *strumieniem danych wewnętrznych.*

Instrukcje

```
100  DATA  527, 3, -10
110  DATA  527, 0,   7
150  '...
200  DATA  7,"alfabet"
210  DATA  5,"BASIC"
```

tworzą strumień danych wewnętrznych złożony z wartości:

```
527 , 3 ,-10 , 527 , 0 , 7 , 7 , alfabet , 5 , BASIC
```

Wprowadzenie wartości ze strumienia do odpowiednich zmiennych programu jest dokonywane za pomocą *instrukcji wczytywania.*

READ *lista_zmiennych*

Na *liście_zmiennych* są umieszczone nazwy zmiennych, oddzielone przecinkami, którym mają być przypisane kolejne wartości ze strumienia danych wewnętrznych.

Ze strumieniem danych jest związany *wskaźnik wprowadzania* kolejnych wartości, który po uruchomieniu programu jest ustawiony na początku strumienia. Wykonanie instrukcji wczytywania powoduje przypisywanie zmiennej z *listy_zmiennych* wartości wskazanej w strumieniu, a następnie przesunięcie

wskaźnika na kolejną wartość. Oczywiście i tutaj musi być zgodność typu zmiennej z typem aktualnie przypisywanej wartości.

Dla utworzonego wyżej strumienia danych wewnętrznych, wykonanie instrukcji

```
50  READ  a,x
60  READ  c,y
70  READ  d,x
80  READ  b1,b$
90  READ  z1,z$
```

spowoduje wprowadzenie wartości do wymienionych na listach zmiennych (rys. 4.4). Należy tutaj zwrócić uwagę na to, że w linii 70 powtórnie wczytano nową wartość zmiennej *x*, zamazując poprzednią.

Wskaźnik wprowadzania danych ze strumienia jest ustawiany z powrotem na pierwszą wartość w strumieniu przy powtórnym uruchomieniu programu lub za pomocą *instrukcji przewinięcia strumienia.*

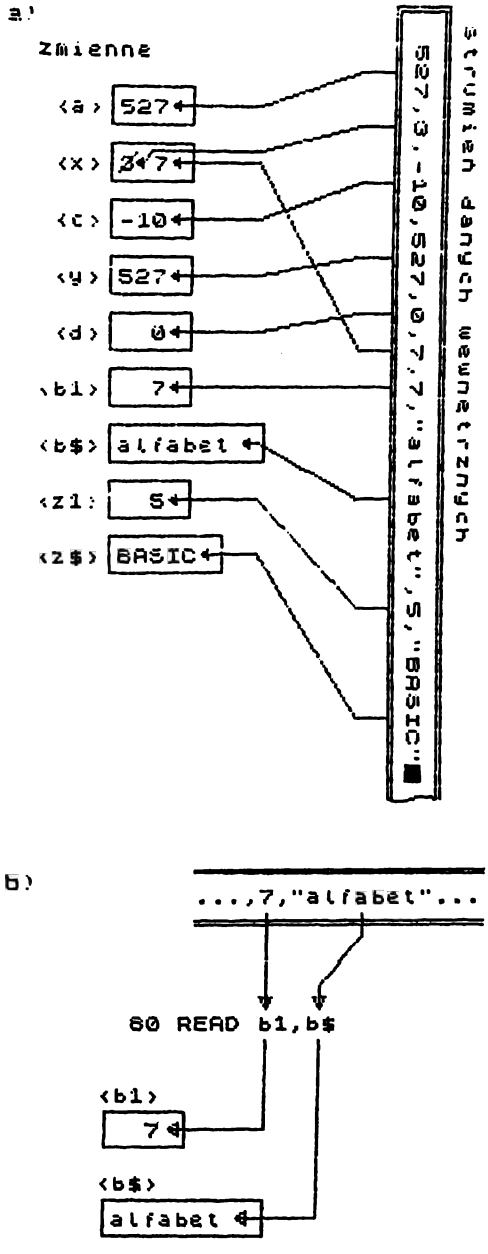
RESTORE
RESTORE *numer_linii*

Następuje przesunięcie wskaźnika wprowadzania danych na początek strumienia danych wewnętrznych. W drugiej postaci instrukcji, wskaźnik wprowadzania danych zostanie przesunięty na element strumienia danych określony *numerem_linii* instrukcji DATA.

Podany poprzednio sposób wprowadzania danych nie zawsze jest wygodny, gdyż wymaga umieszczenia zestawu danych bezpośrednio w programie. Do wczytywania danych z klawiatury służy *instrukcja wprowadzania.*

INPUT *lista_zmiennych*
INPUT "*tekst_zachęty*"; *lista_zmiennych*

Na *liście_zmiennych* są umieszczane nazwy zmiennych oddzielone przecinkami. Strumień danych wejściowych jest tworzony na żądanie programu przez wprowadzenie kolejnych wartości z klawiatury. Wykonanie instrukcji wprowadzania powoduje wstrzymanie działania programu i wyświetlenie znaku pytajnika. *Tekst_zachęty*, ujęty w cudzysłowy, jest wyprowadzony bezpośrednio przed znakiem pytajnika. Znak ten oznacza żądanie wprowadzenia linii zawierającej tyle wartości, ile jest zmiennych na *liście_zmiennych.* Musi być przy tym zachowana zgodność typów wartości z typem odpowiadającej im zmiennej. Po zakończeniu linii znakiem RETURN następuje przypisanie kolejnym zmiennym z listy kolejnych wartości z utworzonego strumienia danych wejściowych.


a)
Zmienne
<a> 527
<x> 7
<c> -10
<y> 527
<d> 0
<b1> 7
<b$> alfabet
<z1> 5
<z$> BASIC
strumień danych wewnętrznych
527,0,-10,527,0,7,7,"alfabet",5,"BASIC"
b)
...,7,"alfabet"...
80 READ b1,b$
<b1>
7
<b$>
alfabet

Rys. 4.4

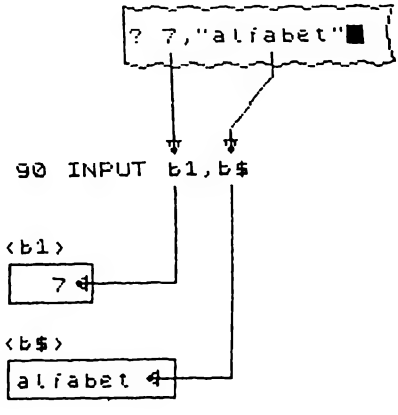
Wykonanie instrukcji

```
80  INPUT  a,x,z
90  INPUT  b1,b$
100 INPUT  "Podaj dlugosc tekstu i tekst"; z1, z$
```

spowoduje wykonanie następujących czynności przez system BASIC:

80 – wstrzymanie działania programu i wyświetlenie symbolu pytajnika
? 37, 21, 75
po którym użytkownik wprowadził trzy wartości, po czym po wprowadzeniu znaku RETURN nastąpiło przeniesienie tych wartości ze strumienia w odpowiednie zmienne i wznowienie działania programu.

90 – wstrzymanie działania programu i wyświetlenie symbolu pytajnika
? 7, "alfabet".
po czym po wprowadzeniu linii z wartościami nastąpiło przeniesienie tych wartości ze strumienia w zmienne (rys. 4.5) i wznowienie programu.



Rys. 4.5

100 – wstrzymanie działania programu i wyświetlenie tekstu, zakończonego symbolem pytajnika:
Podaj długość tekstu i tekst ? 5, "BASIC"
po czym są dokonywane czynności jak dla linii numer 90.

4.6. Instrukcja wyprowadzania wyników

Celem wykonania programu jest uzyskanie wyników obliczeń dla zadanego zestawu danych. Do zaznaczenia w programie miejsca wyprowadzania wartości zmiennych zawierających wyniki służy *instrukcja wyprowadzania*. Wartości mogą być wyświetlane na ekranie lub drukowane na papierze drukarki.

PRINT *lista_wartości*
? *lista_wartości*
LPRINT *lista_wartości*

Na *liście_wartości* są umieszczone stałe oraz nazwy zmiennych, których wartości mają być wyprowadzone.
Poszczególne elementy *listy_wartości* są oddzielane średnikami, spacjami lub przecinkami (wspólnie zwanymi separatorami).
Wykonanie instrukcji polega na dołączeniu do strumienia wyjściowego kolejnych wartości z listy jako ciągu znaków. Separatory na liście określają też rozmieszczenie wartości w strumieniu.
Instrukcja PRINT oraz ? wyprowadza na bieżąco zawartość strumienia wyjściowego na ekran, a instrukcja LPRINT – na papier drukarki.

Oznaczenie instrukcji wyprowadzania symbolem znaku pytajnika jest szczególnie użyteczne dla trybu bezpośredniego wykorzystywania systemu BASIC. Wyprowadzany strumień wyjściowy jest wypełniany znakową postacią wartości, a więc muszą być one przetworzone z postaci wewnętrznej w odpowiedni ciąg znaków. Wartości typu tekstowego są wyprowadzone jako ciąg znaków bez cudzysłowów. Dla wartości typu liczbowego jest dobierana najodpowiedniejsza postać stałej liczbowej odwzorowującej tę wartość. Wartości liczbowe są wyprowadzone z dokładnością do 6 cyfr znaczących. Wartości dodatnie są poprzedzone znakiem odstępu (spacji) oraz po każdej liczbie jest zawsze wyprowadzany jeden odstęp.

Rozmieszczanie wartości w wierszu określają separatory z listy wartości według następujących zasad:

1. Średnik lub spacja powoduje łączenie kolejno wyprowadzanej wartości.
2. Przecinek powoduje wyprowadzenie kolejnej wartości od początku kolejnej strefy (14-znakowej).
3. Każda instrukcja wyprowadzania wyprowadza pojedynczy wiersz. Brak *listy_wartości* powoduje wyprowadzenie wiersza pustego.
4. Zakończenie *listy_wartości* średnikiem lub przecinkiem powoduje złączenie z nią *listy_wartości* z następnej wykonywanej instrukcji wyprowadzania.

Dla ilustracji użycia średnika lub przecinka jako separatora, sprawdźmy w działaniu kilka przykładów. Najpierw w trybie bezpośrednim, gdzie wprowadzone teksty są bądź złączone, bądź rozsuwane od siebie. Podobnie są wyprowadzane wartości liczbowe.

```
Ok
? "Krzys" ; "Iszkowski"
KrzysIszkowski
Ok
? "Krzys" ; " Iszkowski"
Krzys Iszkowski
Ok
? "Krzys" , "Iszkowski"
Krzys          Iszkowski
Ok
? 2+3 ; "=" ; 5
 5 = 5
Ok
? 2+3 , "=" , 5
 5             =             5
Ok
```

Kolejny przykład zawiera praktycznie tylko instrukcje wyprowadzania. Wynik działania tego programu, wyświetlany na ekranie jest pokazany jako wydruk. Sekwencje liczb (wyprowadzonych jako napis) pozwalają zorientować się o pozycji rozmieszczanych znaków reprezentujących wartości.

```
100  LET  x=1.234567
110  LET a1=-80
120  LET a$="BASIC"
130
140  PRINT  "------------"
150  PRINT  5
160  PRINT  a$
170  PRINT  a1
180  PRINT  x
190  PRINT  "............"
195  PRINT
200  PRINT  "12345678901234567890123456789O"
210  PRINT  "^" ; "^"
220  PRINT  a$ ; a1
230  PRINT  "------------"
240  PRINT  "x =" ; x
250  PRINT  5 ; x
260  PRINT
300  PRINT  "12345678901234567890123456789O"
310  PRINT  "^" , "^"
320  PRINT  a$ , a1
330  PRINT  "------------------------------"
340  PRINT  "x =" , x
350  PRINT  5 , x
360
```

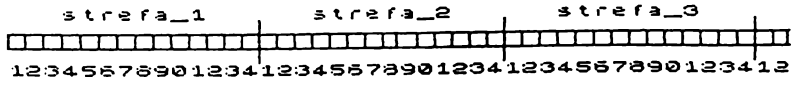
```
RUN
------------
 5
BASIC
-80
 1.234567
............

12345678901234567890123456789O
^^
BASIC-80
------------
*  = 1.234567
 5  1.234567

12345678901234567890123456789O
^                ^
BASIC            -80
--------------------------------
*  =              1.234567
 5                1.234567
Ok
```

Wykonanie każdej instrukcji wyprowadzania powoduje wyprowadzenie ciągu znaków reprezentujących wartości z jej listy oraz znaku zmiany wiersza. Zakończenie *listy_wartości* średnikiem lub przecinkiem wstrzymuje wprowadzanie do strumienia znaku zmiany wiersza, co umożliwia utworzenie jednego wiersza z kilku list wartości.

Następny przykład

```
LIST
100  PRINT  "12345678901234567890"
110  PRINT  "#"
120  PRINT  "#" ;
130  PRINT  "#"
140  PRINT  "#" , "#"
150  PRINT
160  PRINT  "#"
Ok
RUN
12345678901234567890
#
##
#               #

#
Ok
```

Do wyjaśnienia zasady rozmieszczania wartości przy wykorzystaniu przecinka jako separatora, wprowadzimy pojęcie *strefy*. Strefą będziemy nazywać kolejny ciąg 14 (czasem jest to inna liczba) znaków licząc od początku wiersza (rys. 4.6). Przykładowo więc, wiersz ekranu liczący 70 znaków jest logicznie podzielony



Rys. 4.6

na 6 stref (ostatnia strefa zawiera tylko 10 znaków). Używając przecinka jako separatora możemy wyprowadzać wyniki w kolumnach, dla łatwiejszej ich analizy.

```
LIST
100  PRINT  1
110  PRINT  20
120  PRINT  300
130  PRINT  4000
140  PRINT  50000!
150  PRINT  50000! ; 1
160  PRINT  50000! ; 20
170  PRINT  50000! ; 300
180  PRINT  50000! ; 4000
190  PRINT  50000! ; 50000!
200  PRINT  50000! , 1
210  PRINT  50000! , 20
220  PRINT  50000! , 300
230  PRINT  50000! , 4000
240  PRINT  50000! , 50000!
Ok
RUN
 1
 20
 300
 4000
 50000
 50000  1
 50000  20
 50000  300
 50000  4000
 50000  50000
 50000          1
 50000          20
 50000          300
 50000          4000
 50000          50000
Ok
```

Jedyną wadą tego prostego formatowania postaci wyprowadzanej, jest rozmieszczenie wartości z lewostronnym ich wyrównywaniem — co nieco przeczy znanym nam ze szkoły zasadom zapisu rachunków. Ale omówione konstrukcje nie wyczerpują wszystkich dostępnych możliwości — a dla naszych potrzeb są na razie wystarczające.

4.7. Instrukcja komentarza

Kodowanie algorytmu w języku programowania powoduje uzyskanie formalnego zapisu czynności, jakie ma wykonać komputer. Zapis formalny przygotowany zgodnie z wymaganiami składni języka jest zrozumiały dla komputera, ale staje się mało czytelny dla człowieka. Dla zwiększenia czytelności tekstu programu wykorzystuje się *instrukcje komentarza.*

REM *dowolny_tekst*
' *dowolny_tekst*

Umieszczenie w instrukcji *dowolnego_tekstu* może być opisem czynności wykonywanych przez kolejne instrukcje programu. Tekst ten nie jest analizowany przez translator języka.
W drugiej postaci, instrukcja komentarza może być umieszczona w linii za inną instrukcją.

Nagłówek pewnego programu może być następujący:

```
5   REM  Program obliczania wartosci sredniej
6   REM  ------------------------------------
7   REM  Autor             : Waclaw Iszkowski
8   REM  Data napisania    : 85/06/28
9   REM  Data aktualizacji: 85/11/18
10  REM  @Copyright        : WNT,Poland
11  REM  ------------------------------------
12  '  Opis algorytmu
13  '     Program wyznacza wartosc srednia
14  '     dla wartosci wprowadzanych z kla-
15  '     wiatury.
16  '     Koncem zestawu wartosci danych
17  '     jest wartosc  0.
18  '  Opis zmiennych:
19  '     x  -kolejna wartosc wprowadzana,
20  '     n  -liczba wartosci wprowadzonych,
21  '     s  -suma wartosci wprowadzonych,
22  '     w  -wartosc srednia.
23  '---------------------------------------
```

Niestety, w programach zamieszczonych w książce, ze względu na oszczędność miejsca będę te komentarze skracał, ale bardzo namawiam do ich stosowania.

4.8. Instrukcje zakończenia i wstrzymania

Wszystko ma swój koniec. Również program powinien się kończyć. Właściwość ta jest podstawowym elementem w poszukiwaniu odpowiedzi na pytanie — czy dany program jest poprawny. Program działający bez końca — zapętlony — jest programem błędnym.

END

Wykonanie tej instrukcji powoduje zakończenie działania programu i przełączenie systemu BASIC na poziom wprowadzania poleceń.

Instrukcja zakończenia programu może się znajdować w dowolnym jego miejscu. Zwykle jet ona umieszczona po linii programu o najwyższym numerze. Po

zakończeniu wykonywania programu możemy go uruchomić od początku zleceniem RUN. Zlecenie to równocześnie nadaje wszystkim zmiennym typu liczbowego wartość równą 0, a tekstowego — wartość napisu pustego.

Do chwilowego zatrzymania wykonywania programu służy *instrukcja wstrzymania.*

STOP

Wykonanie instrukcji powoduje wstrzymanie działania programu i wyprowadzenie na ekran komunikatu[1]

Break in line *numer_linii*

Ok

z przełączeniem systemu BASIC na poziom wprowadzania poleceń.

[1] Break in line... — Wstrzymanie w linii.

Program wstrzymany z jego wnętrza wykonaniem instrukcji wstrzymania STOP można uruchomić od początku zleceniem RUN lub wznowić *zleceniem wznowienia* CONT.

CONT

Żądanie wznowienia wstrzymanego programu od następnej linii po linii z instrukcją STOP. Zlecenie to może być wydane jedynie wówczas, gdy została uprzednio wykonana instrukcja STOP.

Zastosowanie tej pary: instrukcji STOP i zlecenia CONT, będzie pokazane w rozdz. 10.

Styl programowania

Wprowadzony w tym rozdziale język Mikro BASIC[1] nie pozwala jeszcze na zbyt wiele. Jednak już od początku poznawania języka należy zwrócić uwagę na następujące sprawy.

Dobór identyfikatorów. Ograniczona (na razie) możliwość wyboru identyfikatorów nazw zmiennych nie może zwolnić od przemyślanego ich doboru dla poszczególnych zmiennych. Należy bowiem zauważyć, że zmienne w programie odwzorowują rzeczy istniejące w specyfikacji problemu. I tak identyfikatory nazw zmiennych typu liczbowego powinny być przykładowo kojarzone z następującymi rzeczami:

[1] Nazwa języka Mikro BASIC powstała jedynie na potrzeby tego rozdziału.

a, b, c,	– współczynniki funkcji, formuł;
h	– wysokość, krok obliczeń;
i, j, k	– indeksy;
l, m, n	– liczba elementów;
o	– lepiej nie używać, bo się może pomylić z cyfrą 0;
pi	– stała równa 3.141593;
r	– promień;
s	– suma;
t	– czas;
w	– wartość, omega;
x, y, z	– argumenty, zmienne funkcji i wyrażeń.

Podobna zasada powinna obowiązywać dla identyfikatorów nazw zmiennych typu tekstowego.

Zestaw danych. Instrukcje danych mogą być rozrzucone po całym programie. Jednak dla wygody weryfikacji ich poprawności i liczności oraz ewentualnej wymiany należy zgrupować je w jednym miejscu, starając się też zapisać je tak, aby pojedynczej instrukcji wczytywania z kilkoma zmiennymi na liście odpowiadała jedna instrukcja danych.

Rzetelność danych. Podstawowym warunkiem uzyskania prawdziwych wyników dla danego zbioru danych jest dostarczenie ich programowi dokładnie według założeń algorytmu. Dobrą zasadą jest więc programowe sprawdzenie, co do zakresu wartości wprowadzonych danych oraz wyprowadzenie ich bezpośrednio po wczytaniu, dla kontroli przez użytkownika programu.

Komentarze. Program bez komentarzy, a więc bez dokumentacji jest nic nie wart. Zdanie to powtarzają sobie dyrektorzy firm opracowujących oprogramowanie, w których po odejściu programisty z pracy zostawały sterty wydruków programów bez komentarzy. Programów tych nikt nie był w stanie poprawić, czy nawet tylko wykorzystać.

Konieczne jest więc przyzwyczajenie się do komentowania programu, już na etapie jego wczesnego kodowania. W komentarzach należy umieszczać najistotniejsze uwagi umożliwiające zrozumienie idei zapisanego w języku programowania algorytmu. Pożądanym jest opisanie przeznaczenia większości istotnych zmiennych, postaci strumienia wejściowego danych oraz wyróżnienie logicznie domkniętych części (modułów) programu. A czego nie należy komentować – bo i tutaj przesada jest również szkodliwa. Podam tylko prosty przykład, bo trudno jest opisać wszystkie „genialne" pomysły. A więc

```
37  REM  zmienna  v  przyjmuje wartosc zmiennej  z
40  LET  v=z
```

komentarz w linii jest zbędny, chyba że ma być to przykład programu dla studiujących dopiero elementy języka.

Struktura programu. Tekst programu powinien być wyraźnie podzielony na części (moduły) wykonujące logicznie odrębne fragmenty algorytmu. Podział ten może być dokonany poprzez rozróżnienie poszczególnych części komentarzami i innym zakresem zmienności numerów linii.

Podsumowaniem zaprezentowanych elementów stylu programowania jest prosty program.

```
5   REM     Malo skomplikowany program
10  REM     -------------------------
14  '
15  '     Program wymienia miejscami po
16  '     jednej wartosci z trojek zmien-
17  '     nych wczytywanych z wewnetrzne-
18  '     go zestawu danych z trojka war-
19  '     tosci wprowadzanych z klawiatu-
20  '     ry.
21  '
22  '
50  REM   Inicjacja zmiennych
60  LET   s=0
70  '
100 REM   Wprowadzanie danych
110 READ  a,b,c
120 INPUT x,y,z
130 PRINT "DANE:"
140 PRINT "abc =";a,b,c
150 PRINT "xyz =";x,y,z
160 '
200 REM   Obliczenia programu
202 '     - tutaj trywialne
210 LET   s=b
220 LET   b=y
230 LET   y=s
240 '
300 REM   Wyprowadzanie wynikow
310 PRINT
320 PRINT "WYNIKI:"
330 PRINT "abc =";a,b,c
340 PRINT "xyz =";x,y,z
350 PRINT "s   =";s
360 '
500 REM   Zestaw danych
510 DATA  125 , 321 , 675
520 '
999 END
```

Sytuacje błędne

Już nawet w języku Mikro BASIC jest możliwe popełnienie błędów. Oczywiście pamiętamy tutaj nadal, że linie o postaci niezgodnej ze składnią instrukcji oraz polecenia spowodują wystąpienie błędu składni.

? Redo from start, Retype line

? Ponownie od początku. Przepisz linię. Komunikat ten pojawia się w przypadku, gdy w linii zawierającej strumień danych wyjściowych dla instrukcji wprowadzania INPUT popełnimy błąd. Danych tych było za mało lub poszczególne typy wartości były niezgodne z typami odpowiadających im zmiennych.

Często być może brakło znaku cudzysłowu zamykającego napis, a początkujący w programowaniu, zamiast znaku kropki dziesiętnej wstawili znak przecinka.

Po tym komunikacie należy spokojnie jeszcze raz wprowadzić linię z danymi, sprawdzając ewentualnie w tekście programu, czy może czegoś tam nie brakuje lub czy przez pomyłkę nie jest za dużo zmiennych. Wartości zostaną wstawione w zmienne dopiero wówczas, gdy wszystkie będą poprawne.

Out of data

Za mało danych. Przyczyna tego komunikatu jest podobna jak poprzedniego, przy czym odnosi się on do instrukcji wczytywania danych ze strumienia danych wewnętrznych (READ i DATA). Tutaj po sprawdzeniu zgodności typów zmiennych i wartości oraz ich liczności i po poprawieniu błędu musimy program uruchomić od początku. Aha – w zestawie danych może być więcej niż potrzeba wartości – tylko po co?

Przy okazji warto zwrócić uwagę na to, że pisząc program często nie wiemy dla ilu danych będzie on wykonywany za każdym razem. Dlatego też musimy wtedy zaprojektować wprowadzenie danych, tak aby:

– na początku zestawu znajdowała się liczba określająca liczność danych lub
– na końcu zestawu znajdowała się charakterystyczna, łatwa do wykrycia wartość – na przykład 0 lub 999999 – nie należąca do zbioru danych.

Do zagadnień tych powrócimy w rozdziale 6.

Type mismatch

Niezgodność typów wystąpi w instrukcji przypisania, jeżeli zmiennej przypisujemy wartość innego typu. Inaczej mówiąc zmiennej typu liczbowego przypisaliśmy wartość typu tekstowego lub odwrotnie. Ewentualnie w zapisie zmiennej, która miała być typu tekstowego, zapomnieliśmy o symbolu dolara $.

Stop at line *numer_linii*

Zapętlenie. Program po uruchomieniu działa i działa, i nic się nie dzieje. Wyników bądź nie ma, bądź też są tylko częściowe, a my już tracimy cierpliwość. Jeżeli nie zadaliśmy świadomie lub nieświadomie na przykład obliczeń wymagających milionów lub nawet tylko tysięcy operacji – co w sposób naturalny zajęło czas komputera – program należy przerwać znakiem sterującym CTRL-C. System BASIC zostanie wtedy przełączony na poziom wprowadzania poleceń i zostanie wyprowadzony komunikat wskazujący *numer_linii*, w której nastąpiło przerwanie działania.

Teraz w trybie bezpośrednim możemy wyprowadzić zawartość wybranych zmiennych i spróbować określić przyczynę takiego długotrwałego działania programu. A może my byliśmy zbyt niecierpliwi, gdyż po prostu mamy za wolny komputer jak na nasze potrzeby.

Do tych problemów powrócę w rozdziale 10.

Problemy

4.1. Podane niżej wartości zapisz w poprawnej postaci stałych typu liczbowego (a...d) oraz typu tekstowego (e...h)

a) 1.257
b) 1/3
c) 10^7
d) zero
e) 1,257
f) BASIC
g) swoje nazwisko
h) 10 spacji

4.2. Podaj jakie wartości będą miały zmienne po wykonaniu fragmentów programów:

```
a)  10 READ  x,y,z
    20 READ  a,b,c
    30 DATA  -67.25, 1E-10, 8.43
    40 DATA   8.275,      3     5
```

```
b)  10 READ  i$,n$,p$
    20 READ  nl$,nr,pk
    30 DATA  Waclaw, Iszkowski
    40 DATA  "Instytut Informatyki"
    50 DATA  Nowowiejska, 19, 301
```

```
c)  10 INPUT  i,j,k,l$
    dla strumienia wejsciowego
    ? 6,7,9,KONIEC
```

```
d)  10 INPUT  a$,b$,c$,d$
    dla strumienia wejsciowego
    ? beczka smiechu, cyrk, "102"
```

4.3. Podaj postać strumienia wyjściowego po wykonaniu podanych fragmentów programu

```
a) 10  PRINT  "2  razy  2  = 2"; 4 ;" za dokladnie..."
```

```
b) 10  LET    x=5
   20  PRINT  "X ="; x ;" czyli  X+X ="; 10
   30  PRINT  "X ="; x ;" czyli  X*X ="; 25
```

```
c) 50  PRINT  1.27E-5, 52E+3, 0
   60  PRINT  1E+6    , 1E+7
```

```
d)  90  LET    t$="w jezyku BASIC -"
   100  PRINT  "Programowanie "; t$ ; 80
   110  PRINT  "Nauka programowania "; t$ ;" ZX Spectrum"
```

4.4. Podaj jakie błędy zostały popełnione w podanych przykładach zapisu elementów języka

```
a)  1,527.35
b)  10  LET  x="alfa"
c)  20  PRINT "xy="; xy ;"zl
d)  30  READ  "delta="; d, e, f, g
e)  40  la$= +0
f)  50  DATA 5,6,x,10,gamma", mi$
g)  60  INPUTA  "Dane"; c
```

Let's learn BASIC English

letter litera; **capital** ~, **upper** ~ duża litera; **small** ~, **lower** ~ mała litera (upper i lower pochodzi od sposobu przełączania wprowadzania znaków z klawiatury na górnej i dolnej bazie)
digit cyfra; **significant** ~ cyfra znacząca
type typ
constant stała
variable zmienna
numeric liczbowy (czasem też używamy) numeryczny; ~ **type** typ liczbowy; ~ **constant** stała typu liczbowego, stała liczbowa; ~ **variable** zmienna typu liczbowego, zmienna liczbowa
string napis (czasem też używamy łańcuch, tekst); ~ **type** typ tekstowy; ~ **constant** stała typu tekstowego, stała tekstowa; ~ **variable** zmienna typu tekstowego, zmienna tekstowa (tutaj rozróżniamy postać zewnętrzną wartości – jest nią napis, od typu tej wartości, który nazywamy typem tekstowym)
integer całkowity
integer number liczba całkowita, liczba typu całkowitego
real rzeczywisty
real number liczba rzeczywista, liczba ułamkowa, liczba typu rzeczywistego
fraction ułamek
mantissa mantysa
exponent wykładnik
scientific notation liczba wykładnicza; liczba w postaci wykładniczej złożona z mantysy i wykładnika
LET NIECH, PRZYPISZ, OBLICZ
LET **a** = **b** – czytamy „zmiennej *a* przypisz wartość zmiennej *b*"
READ CZYTAJ z wewnętrznego strumienia danych
DATA dana, dane
INPUT WPROWADŹ ze strumienia wyjściowego
PRINT WYŚWIETL – tłumaczenie dokładnie według realizowanej czynności; DRUKUJ – tłumaczenie związane z dawnym wyprowadzeniem wartości na papier dalekopisu (specjalnej maszyny do pisania); WYPROWADŹ do strumienia wyjściowego
LPRINT (**Lineprinter PRINT**) WYDRUKUJ na drukarce
REMark UWAGA, komentarz
END KONIEC; zakończ wykonywanie programu
STOP STOP; wstrzymaj wykonywanie programu
CONTinue KONTynuuj; wznów wykonywanie programu
character znak (pisarski); **control** ~ znak sterujący; **graphic** ~ znak graficzny
blank, space odstęp, spacja
= **egual sign, assigment symbol** znak równości, symbol przypisania
\+ **plus sign** znak liczby dodatniej, dodawania
– **minus sign** znak liczby ujemnej, odejmowania
* **asterisk, multiplication symbol** gwiazda, gwiazdka, symbol mnożenia
/ **slash, division symbol** kreska ukośna, symbol dzielenia
↑ **up arrow, exponentation symbol** górna strzałka, symbol potęgowania, grot
(**left parenthesis** lewy nawias okrągły
) **right parenthesis** prawy nawias okrągły
% **percent** procent
\# **sharp, hash, pound symbol** krzyżyk, symbol funta (tutaj czasem występuje £)
$ **dolar symbol** symbol dolara (tutaj czasem występuje ¤ – symbol waluty)

[left bracket lewy nawias kwadratowy
] right bracket prawy nawias kwadratowy
, comma przecinek
. period, decimal point kropka, kropka dziesiętna
' single guotation mark, apostrophe apostrof
: colon dwukropek
& ampersand ampersend, handlowe "i"
? question mark pytajnik
< less than mniejsze niż
> greater than większe niż
\ backslash, integer division symbol lewa kreska ukośna, symbol dzielenia całkowitego, ukośnik
@ at-sign, commercial at handlowe "przy" (studenci nazywają go embrionem lub małpą)

END of lesson.

5. Obliczenia numeryczne

Pierwotną i podstawową funkcją komputera jest wykonywanie obliczeń numerycznych na wartościach typu liczbowego. W tym zakresie działania, komputer jest podobny do kalkulatora. Jego przewaga polega na możliwości wcześniejszego zapisu formuły obliczeń i następnie na wielokrotnym jej wykorzystywaniu dla kolejnych zbiorów danych.

W tym rozdziale zajmiemy się wyjaśnieniem składni wyrażenia oraz metodami jego obliczania. Omówimy też wykaz funkcji standardowych.

5.1. Wyrażenia arytmetyczne

Język wysokiego poziomu umożliwia zbiorcze zapisanie obliczenia numerycznego w formie wyrażenia arytmetycznego znanego w matematyce. W stosunku do zapisu matematycznego, wyrażenie w języku programowania jest zapisywane w pojedynczej linii z dokładnie określoną kolejnością obliczeń.

Wyrażenie arytmetyczne jest formułą złożoną z argumentów oraz operatorów arytmetycznych.

Argumentem wyrażenia może być stała, zmienna, funkcja oraz wyrażenie ujęte w nawiasy okrągłe. *Wyrażeniami prostymi* są pojedyncze argumenty

2.751 −5 c −28 SIN(X)

Operatorami arytmetycznymi, w kolejności malejących priorytetów, są:

↑ potęgowanie	(7)
− zmiana znaku	(6)
* / mnożenie, dzielenie	(5)
+ − dodawanie, odejmowanie	(4)

Zapis wyrażenia.

1. W wyrażeniu mogą występować jedynie argumenty i operatory typu liczbowego oraz nawiasy okrągłe.
2. Wszystkie operatory muszą być pisane w sposób jawny.
3. Między każdymi dwoma argumentami musi wystąpić operator i między każdymi dwoma operatorami musi wystąpić argument lub lewy nawias.
4. Lewe nawiasy okrągłe, jeżeli występują, muszą bezpośrednio poprzedzać argument, inny lewy nawias lub operator negacji, a prawe nawiasy muszą wtedy następować bezpośrednio za argumentem lub innym prawym nawiasem. Liczba lewych nawiasów w wyrażeniu musi być równa liczbie prawych. Dla każdej pary nawiasów: lewy musi poprzedzać w wyrażeniu prawy.

Przykłady wyrażeń zapisanych poprawnie:

```
x + y * 3.6E-5
-3.27 * r ^ 2
a * (b-a) / ((a+b) * (a-b))
k * (-n) + 1^(-n)
```

Przykłady wyrażeń zapisanych błędnie (w komentarzu podano numer naruszonej zasady):

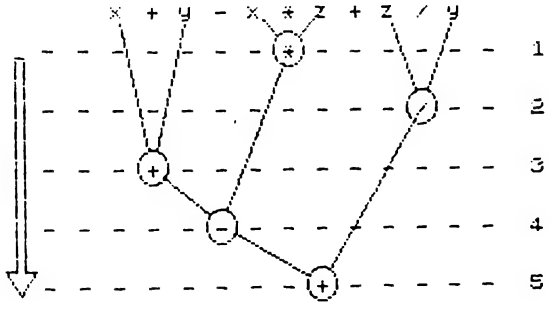
a + s$/c	– (1), s$ nie jest typu liczbowego;
(a + b)(a – b)	– (2), znaku operacji mnożenia nie wolno opuszczać;
5.62 a – z	– (3), między argumentami musi wystąpić operator;
x* – 85.7	– (3), między operatorami * i – musi wystąpić nawias;
–(x – 5)	– (4), operator – nie jest tutaj operatorem negacji, a oznacza odejmowanie;
((e + f)/(g + f)	– (4), niezgodność liczby nawiasów;
x)*(y	– (4), w parze nawiasów, lewy nie poprzedza prawego.

Kolejność obliczania wartości wyrażenia.

1. Obliczanie jest wykonywane od najbardziej zagnieżdżonego w nawiasach wyrażenia.

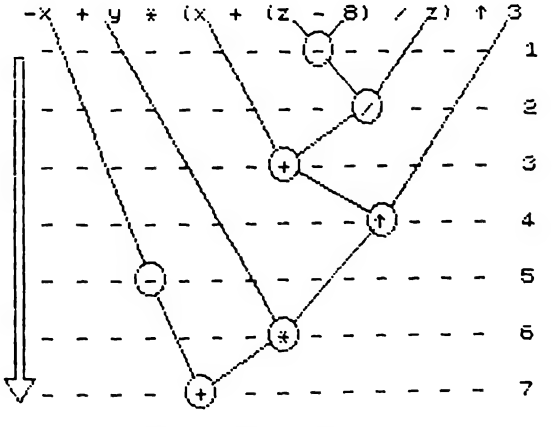
2. W części wyrażenia ograniczonej nawiasami (lub dla całego wyrażenia) są wykonywane kolejno operacje o zmniejszającym się priorytecie.
3. W przypadku równych priorytetów pierwszeństwo mają operacje występujące w kolejności od strony lewej do prawej.

Zgodnie z tymi zasadami, kolejność obliczania wartości dla przykładowych wyrażeń jest pokazana na rys. 5.1 i 5.2.



Rys. 5.1

Dla obliczenia z rys. 5.1 są wykonywane najpierw kolejno od lewej strony operacje * i / o równych priorytetach, następnie znowu kolejno od lewej strony operacje +, − i + o równych priorytetach.



Rys. 5.2

Dla obliczenia z rys. 5.2 – najpierw wyszukujemy najbardziej zagnieżdżoną w nawiasach część wyrażenia i wykonujemy operację −, po czym / i + w kolejnej części wyrażenia w nawiasach; następnie najwyższy priorytet ma operacja ↑, potem negacja −, i dalej operacja *, a na końcu operacja +.

Teraz na podstawie zasad zapisu wyrażeń i kolejności obliczeń możemy pokazać przykłady formuł matematycznych zapisanych w języku BASIC.

b^2-4ac	`b ^ 2 - 4 * a * c`
$\frac{y-b}{x-a}$	`(y - b) / (x - a)`
$\frac{1}{1+x^2}$	`1 / (1 + x^2)`
$A+\frac{BC}{D^T}$	`a + b*c / d^2`
$x-\frac{y}{xz}$	`x - y / (x * z)` `x - y / x / z`
$y'+3y''-2y'''$	`y1 + 3 * y2 - 2 * y3`
$\sqrt[3]{\frac{1}{x^2+2x+1}}$	`(1 / (x * (x+2) + 1)) ^ (1 / 3 )`
$\cfrac{\alpha}{\beta+\cfrac{\alpha}{\beta+\cfrac{\alpha}{\gamma}}}$	`a / (b + a / (b + a/d))`
$x^{(1-n)^{-n}}$	`x ^ ((1-n) ^ (-n))`

5.2. Standardowe funkcje arytmetyczne

Przy konstruowaniu wyrażeń wyraźnie brakuje nam pewnych, od dawna znanych w matematyce funkcji. Jednym tchem możemy je wymienić: wartość bezwzględna, pierwiastek, logarytm, funkcje trygonometryczne, itd. Oczywiście zdajemy też sobie sprawę z trudności obliczenia wartości tych funkcji, gdyż wymaga to dla uzyskania odpowiedniej dokładności obliczenia wielu wyrazów rozwinięcia w szereg.

Używając języka BASIC nie musimy sięgać do poradnika matematycznego dla wyszukania metody wyliczenia wartości tych funkcji. Język dostarcza bowiem zestaw standardowych funkcji, które można bezpośrednio wykorzystywać w wyrażeniach.

Wywołanie funkcji standardowej

nazwa_funkcji (*argument*)

Nazwa_funkcji jest trzyliterowym identyfikatorem, a *argument* dowolnym, poprawnie skonstruowanym wyrażeniem typu liczbowego.

W obliczaniu wartości wyrażenia, napotkanie nazwy funkcji powoduje najpierw obliczenie wartości wyrażenia argumentu, a następnie wartości funkcji przez standardowy zapisany w bibliotece systemu podprogram. Obliczona wartość funkcji jest następnie wykorzystywana w wyrażeniu, tak jak wartości zmiennych czy stałych.

Matematyczne funkcje standardowe:

ABS(X) – wartość bezwzględna ($|x|$)
FIX(X) – część całkowita wartości x
INT(X) – część całkowita nie większa od wartości x
SGN(X) – wartość znaku (-1 gdy $x < 0$, 0 gdy $x = 0$, 1 gdy $x > 0$)
EXP(X) – e^x, $x \leqslant 87.3365$
LOG(X) – $\ln x$, $x > 0$
SQR(X) – $\sqrt{x}$, $x \geqslant 0$
ATN(X) – arctg$x$, wynik w radianach w przedziale $(-\pi/2, \pi/2)$
COS(X) – cos$x$, argument w radianach
SIN(X) – sin$x$, argument w radianach
TAN(X) – tg$x$, argument w radianach

Funkcje ABS(X) i SGN(X) umożliwiają proste rozdzielenie wartości argumentu na wartość bezwzględną i wartość znaku.

Stąd dla przykładowych wartości mamy:

X	SGN(X)	ABS(X)
−52.68	−1	52.68
0	0	0
67.251	1	67.251

Funkcje INT(X) i FIX(X) są przeznaczone do podziału wartości na część całkowitą i dziesiętną. Ich działanie rozważmy na przykładzie:

X	INT(X)	FIX(X)
6.951	6	6
5.5	5	5
4.23	4	4
3	3	3
2.11	2	2
0.7	0	0
0	0	0
−0.7	−1	0
−2.11	−3	−2
−3	−3	−3
−4.23	−5	−4
−5.5	−6	−5
−6.951	−7	−6

Teraz możemy jeszcze raz przeczytać opis obu funkcji i porównując go z przykładem przeanalizować ich działanie. O użyteczności tych funkcji możemy się przekonać chcąc dokonać:

a) zaokrąglenia wartości z określoną dokładnością:

INT(x + 0.5) — do najbliższej wartości całkowitej,
INT(x*10 + 0.5)*0.1 — do jednego miejsca dziesiętnego po kropce,
INT(x*100 + 0.5)*0.01 — do dwóch miejsc dziesiętnych po kropce.

b) obcięcia wartości z określoną dokładnością:

FIX(x) — do wartości całkowitej,
INT(x) — do wartości całkowitej (dla wartości dodatnich),
FIX(x*10)*0.1 — do jednego miejsca dziesiętnego po kropce,
FIX(x*100)*0.01 — do dwóch miejsc dziesiętnych po kropce.

c) wyznaczenia wartości reszty z dzielenia zmiennej *a* przez zmienną *n*:

a − INT(a/n)*n — dla *a* i *n* dodatnich,
SGN(a)*(ABS(a) − INT(ABS(a/n))*ABS(n)) — dla *a* i *n* dowolnych.

Funkcje EXP(X) i LOG(X) umożliwiają wyznaczenie innych, pochodnych funkcji jak np.:

0.43429*LOG(x) — $\log_{10} x$, przekształcenie w logarytm dziesiętny,

LOG(x)/LOG(a) – $\log_a x$, przekształcenie w logarytm o dowolnej podstawie a
(EXP(x) – EXPC(–x))*0.5 – $\sinh x$, sinus hiperboliczny,
(LOG(x) + SQR(x*x + 1)) – $\operatorname{arsinh} x$, area sinus.

Funkcje trygonometryczne ATN(X), COS(X), SIN(X), TAN(X) operują na wartości kąta wyrażonej w radianach. Jeżeli mamy więc obliczyć wartość funkcji dla kąta podanego w stopniach, wykorzystujemy przekształcenie

k stopni = $k*\pi/180$ radianów (prościej: $0.017453*k$) i odwrotnie
r radianów = $r*180/\pi$ stopni (prościej: $57.295779*r$)

Korzystając z tych czterech funkcji standardowych możemy wyznaczyć i inne, jak na przykład

1/TAN(x) – $\operatorname{ctg} x$, cotangens,
ATN(x/SQR(–x*x + 1)) – $\arcsin x$, arcus sinus,

oraz inne podobne, również na podstawie ogólnie znanych wzorów.

5.3. Wykorzystywanie wyrażeń arytmetycznych

Podana w poprzednim rozdziale instrukcja przypisania jest też przeznaczona do przypisywania wyniku obliczenia wartości wyrażenia określonej zmiennej.

LET *zmienna = wyrażenie*

Po wykonaniu instrukcji *zmienna*, która musi być typu liczbowego, zawiera wartość *wyrażenia*. W niektórych implementacjach słowo kluczowe LET może być opuszczone.

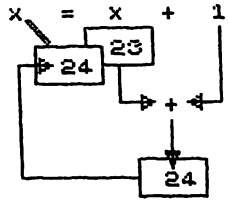
Rozważmy wykonanie instrukcji:

```
5  LET  x = 23
10 LET  x = x + 1
```

Jej wykonanie składa się z następujących czynności:

1. Zapamiętywany jest adres słowa zmiennej stojącej po lewej stronie znaku równości – tutaj adres zmiennej x.
2. Pobierana jest wartość pierwszego argumentu wyrażenia – tutaj wartość zmiennej x (w trakcie analizy wyrażenia stwierdzono, że operacja dodawania ma być wykonana jako pierwsza).
3. Pobierana jest wartość drugiego argumentu wyrażenia – tutaj stała równa 1.
4. Wykonywana jest operacja dodawania.
5. Wynik obliczeń – wartość wyrażenia – jest umieszczony w słowach pamięci wskazanych adresem zapamiętanym w punkcie 1 – a więc w zmiennej x.

Podsumowując, instrukcja zwiększa wartość zmiennej x o 1, co schematycznie pokazuje rys. 5.3.



Rys. 5.3

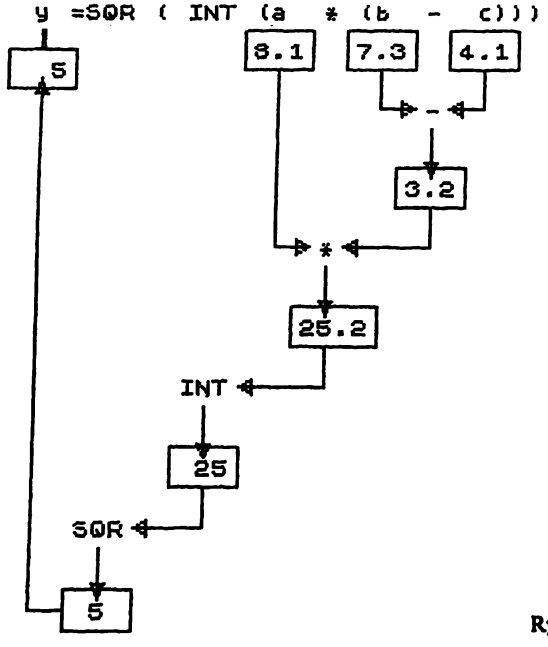
Drugi przykład jest nieco bardziej skomplikowany. Dla formuły

$$y = \sqrt{\text{entier } (a*(b-c))}$$

i wartości $a = 8.1, b = 7.3$ i $c = 4.1$ należy obliczyć jej wartość y. Program w języku BASIC będzie następujący:

```
20  READ  a,b,c
30  LET   y = SQR( INT( a*(b-c)))
40  PRINT y
50  DATA  8.1, 7.3, 4.1
60  END
```

Wykonanie instrukcji przypisania z linii 30 jest zilustrowane na rys. 5.4.



Rys. 5.4

Wyrażenia arytmetyczne można też umieszczać w instrukcji wyprowadzenia jako element listy wartości wyprowadzonych. Obliczona w trakcie wykonania instrukcji wartość jest bezpośrednio wyprowadzana do strumienia wyjściowego. W trybie bezpośrednim (kalkulatorowym) polecenie przypisania i wyprowadzania z wyrażeniami daje możliwość szybkiego obliczenia prostych formuł.

Prześledźmy przykład oparty na obliczeniu formuły z poprzedniego punktu:

```
NEW
Ok
LET  a=8.1
Ok
LET  b=7.3
Ok
LET  c=4.1
Ok
? SQR( INT( a * (b-c) ))
 5
Ok
d=7.3 - 4.1
Ok
a=38
Ok
? SQR( INT( a*d ))
 11
Ok
```

Łatwość korzystania z tego trybu jest zaletą w przypadku chęci szybkiego otrzymania potrzebnych wartości — na przykład wartości funkcji ln 2,75. Polecam tutaj jeszcze raz przejrzenie przykładu z rozdziału 3. Możliwe jest też łatwe przetestowanie zapisu skomplikowanych wyrażeń zanim umieścimy je w tekście programu.

Wyrażenia arytmetyczne, w tym najczęściej ich najprostsza jednoargumentowa postać są też wykorzystywane jako elementy innych instrukcji i zleceń języka. Dla tych potrzeb są użyteczne jedynie wartości całkowite. Stąd też zdefiniujemy tutaj pojęcie *wyrażenia całkowitego.*

Wyrażeniem całkowitym, a dokładniej o wartości całkowitej, jest równoważne mu wyrażenie arytmetyczne, którego wartość zostaje zaokrąglona do wartości całkowitej.

wyrażenie_całkowite = INT(*wyrażenie_arytmetyczne* + .5)

Uwaga: W starszych wersjach implementacji języka BASIC wartością całkowitą wyrażenia arytmetycznego jest tylko część całkowita jego wartości.

Dla ilustracji wykorzystywania wyrażeń arytmetycznych przeanalizujemy przykład prostego programu obliczeń numerycznych.

Przykład 5.1. „Wyznaczenie wysokości strzału pocisku"

Opis problemu

Chcemy wystrzelić pocisk z prędkością początkową v pod kątem α stopni. Pytamy – na jakiej wysokości h będzie się znajdować pocisk w odległości d od wyrzutni? Zakładamy, że prędkość pocisku się nie zmienia.

Metoda rozwiązania

Z fizyki dowiadujemy się, że pocisk osiągnie odległość d po czasie

$$t = \frac{d}{v \cdot \cos\alpha}$$

a wtedy wysokość h pocisku ponad ziemią będzie wynosić

$$h = vt \cdot \sin\alpha - \frac{gt^2}{2}$$

gdzie g = 9.81 m/sek^2 jest stałą grawitacji.

Opis danych

Z klawiatury będziemy wprowadzać kolejno wartości v [m/sek], α [stopnie] i d [m].

Opis wyników

Otrzymamy wartość osiągniętej wysokości h [m].

Tekst programu

```
100  REM  Obliczanie wysokosci strzalu
110  REM  ----------------------------
120  '    Dokumentacja programu w zalaczeniu
130  '
140  '    Opis zmiennych
145  '     p   -stala pi (ludolfina)
150  '     g   -stala grawitacji
155  '     v   -predkosc pocisku
160  '     a   -kat podniesienia
165  '     d   -odleglosc
170  '     h   -poszukiwana wysokosc
175  '     t   -czas
180  '     r   -kat w radianach
190  '
200  REM  Inicjacja stalych
210  LET  p=3.1451
220  LET  g=9.81
230  '
300  REM  Wprowadzanie danych
310  PRINT  "Podaj:"
320  INPUT  " - predkosc [m/sek]                ";v
330  INPUT  " - kat podniesienia [stopnie]      ";a
340  INPUT  " - odleglosc od wyrzutni [m]       ";d
350  '
400  REM  Obliczenia
410  LET  r= a * p/180
420  LET  t= d / ( v * COS(r) )
430  LET  h= t * ( v * sin(r) - g*t*0.5 )
440  '
500  REM  Wyprowadzanie wyniku
510  PRINT
520  PRINT  "Osiagnieto wysokosc = "; h ;" [m]"
600  END
```

Komentarz

W trakcie wykonywania programu może wystąpić błąd dzielenia przez zero w linii 420, jeżeli z zewnątrz podano kąt podniesienia $\alpha = 90°$. Dla niektórych wartości danych (jakich?) może też wystąpić wartość ujemna osiągniętej wysokości.

Styl programowania

Zapis formuł w postaci wyrażeń. Najważniejszą sprawą w zapisie formuł jako wyrażeń arytmetycznych jest znalezienie optymalnego związku między zwiększeniem efektywności obliczania wyrażenia a nie zmniejszaniem jego czytelności.

Zwiększenie efektywności uzyskujemy poprzez następujące posunięcia:

1. Identyczne fragmenty wyrażenia (podwyrażenia) umieszczamy w osobnej, wcześniejszej instrukcji.

   Zamiast

```
10  LET  z=(1+EXP(x)-1/(1+EXP(x)))
```

   lepiej

```
15  LET  ex=1+EXP(x)
20  LET   z=(ex-1/ex)
```

2. Proste obliczenia na wartościach stałych wykonujemy w pamięci własnej lub posługując się trybem kalkulatorowym – wpisując do wyrażenia gotową wartość.

   Zamiast x ↑ (1/2) lepiej x ↑ .5 lub SQR(x)
   y/4 y*.25

3. Czas wykonywania poszczególnych operacji nie jest identyczny. Najdłużej jest wykonywane potęgowanie, nieco krócej dzielenie a potem mnożenie, a najkrócej dodawanie i odejmowanie. Pamiętając o tym należy zastępować operacje dłuższe krótszymi oraz minimalizować liczbę koniecznych do wykonania operacji. Operacja potęgowania dla obliczenia a ↑ b wymaga obliczenia wyrażenia EXP (b*LOG(a)).

   Zamiast x↑3 lepiej x*x*x
   x*2 x+x
   x↑2+x (x+1)*x

   Ten ostatni przykład jest elementem schematu Hornera do efektywnego obliczania wartości wielomianów:

   $x^n + x^{n-1} + \ldots x^2 + x$ obliczamy
   $((x + 1)*x + 1)*x + \ldots + 1)*x$.
   Proszę sprawdzić praktycznie zysk czasowy takiej zamiany.

4. Czas wykonania operacji przez komputer nie zależy od liczby cyfr znaczących i złożoności argumentów. Dla przykładu obliczenie 2*2 jest wykonywane dokładnie tyle samo czasu co obliczenie 67.43528*2.435775. Należy jednak pamiętać, że w systemach BASIC z interpretatorem, do czasu wykonania operacji trzeba doliczyć czas konwersji zewnętrznej postaci liczby do wewnętrznej postaci zmiennoprzecinkowej, a ten jest zależny od liczby cyfr w wewnętrznej postaci w stałej. Stąd też skomplikowane, o dużej liczbie cyfr znaczących i często używane stałe jako współczynniki wyrażeń umieszczamy w osobnych zmiennych.

Z kolei dla zachowania czytelności i zrozumienia wyrażenia należy:

1. Unikać zbytniego przekształcania postaci wyrażenia w stosunku do pierwotnej formuły matematycznej.
2. Wykorzystywać identyfikatory zmiennych podobne do występujących w formule.
3. Stosować nawiasy ujmujące poszczególne podwyrażenia zgodnie z zapisem formuł – np. całe wyrażenie licznika i całe wyrażenie mianownika – oraz uściślające kolejność obliczeń.
4. W zapisie tekstu wyrażenia można zastosować dwie konwencje. Pierwsza z nich unika oddzielania odstępami poszczególnych elementów wyrażenia, druga zaś wprowadza odstępy. Porównajmy więc trzy zapisy – pierwsze dwa uwzględniające podane konwencje, a trzecie będące rozwiązaniem pośrednim.

```
10 LET y=(1/(x*(x+2)+1))^(1/3)
10 LET y = ( 1 / ( x * ( x + 2 ) + 1 ) ) ^ ( 1 / 3 )
10 LET y = ( 1/( x*( x+2 ) + 1 ))^( 1/3 )
```

Porównajmy też inne zapisy

```
10  let x=x+1
10  LET  x = x + 1
10  LET x2 = SQR( X+Y )
10  let xy = log( exp (x-1))
```

Ostateczny dobór postaci zapisu wyrażeń w sytuacji konieczności pogodzenia przeciwstawnych sobie celów pozostawiam programującym. Jedynym wymaganiem jest to, aby był to zapis czytelny i porządny.

Sytuacje błędne

W wykonywaniu obliczeń wartości wyrażeń podstawowy problem istnieje w zapewnieniu odpowiedniej dokładności obliczeń, uniknięciu kumulowania się błędów obliczeń, a przede wszystkim nie przekroczeniu dopuszczalnych zakresów wartości typu liczbowego.

Syntax error

Błąd składni szczególnie często pojawia się dla linii zawierających zapisy wyrażeń. Jego przyczyną jest najczęściej brak jednego z nawiasów okrągłych, zła postać argumentów, itp.

Missing operator

Opuszczony argument jest precyzyjniejszym opisem błędu składni polegającym na braku argumentu po występującym w zapisie wyrażenia symbolu operatora. W obu przypadkach należy przejrzeć dokładnie tekst linii i poprawić go zgodnie z zasadami zapisu wyrażeń.

Illegal function call

Nielegalne wywołanie funkcji jest spowodowane niedopuszczalną dla danej funkcji lub operacji wartością argumentu. Należy sprawdzić, czy:
a) wartość argumentu funkcji LOG nie jest mniejsza lub równa 0,
b) wartość argumentu funkcji SQR nie jest ujemna,
c) operacja potęgowania nie dotyczy podnoszenia do potęgi ułamkowej wartości ujemnej.

Overflow

Nadmiar jest spowodowany wykonaniem obliczenia dającego w wyniku wartość przekraczającą zakres wartości systemu BASIC. (ABS (*wartość*) większe od 1E + 38). W przypadku niedomiaru, a więc przekroczenia wartości minimalnych (ABS (*wartość*) mniejsze od 1E − 38), jest przyjmowana wartość zero i obliczenia są kontynuowane.

Division by zero

Dzielenie przez zero jest wykrywane w trakcie obliczania wyrażenia, jeżeli wartość dzielnika jest równa 0. Wykrycie tego błędu nie powoduje przerwania działania programu, a jako wynik dzielenia jest przyjmowana największa wartość absolutna (1E + 38) ze znakiem wartości dzielnej.

Uwaga: W wielu wersjach języka BASIC dzielenie przez zero powoduje przerwanie wykonywania programu i wyprowadzenie komunikatu. Trzeba tutaj powiedzieć, że bez względu na sposób reakcji systemu BASIC na próbę dzielenia przez zero, powstanie takiej sytuacji świadczy o istnieniu błędu w algorytmie i co więcej jego przyczyna może się znajdować znacznie wcześniej niż to by wynikało z numeru linii, w której wystąpił. Przyczyną tego błędu jest z reguły zły zapis wyrażenia, opuszczenie argumentu lub też brak wcześniejszego nadania wartości początkowej argumentowi będącemu dzielnikiem (proszę pamiętać, że po uruchomieniu programu wszystkie zmienne typu liczbowego mają wartość początkową równą 0).

Problemy

5.1. Zapisz podane formuły matematyczne w postaci wyrażeń w języku BASIC

a) ax^3+bx^2+cx+d

b) $\ln(a^2-2ab\cos x+b^2)$

c) $\dfrac{\cos^2 x}{\sqrt{1-k^2\sin^2 x}}$

d) $\dfrac{x^{a-1}}{1+x^b}$

e) $e^{-a^2x^2}\cos bx$

f) $\dfrac{1}{\delta\sqrt{2\pi}}\exp\left(-\dfrac{(x-m)^2}{2\delta^2}\right)$

g) $n^2(2n^2-1)$

h) $2\log_5(3x-1)-\log_5(12x+1)$

5.2. Podaj wartość wyrażeń

```
a) 1 + 2 + 3 + 4
b) 5 - 6 + 7 - 8
c) 9 * (-13)
d) 50/5*3
e) 128/2/8
f) 128/2^4
```

```
g) (3^2 + 4^2)^(1/2)
h) (17 - 2 )*3
i) 17 - 2 * 3
j) SQR( ABS( INT(-35.6)))
k) SGN( ABS( x+1 ))
l) INT(-14.27) - FIX(-14.27)
```

5.3. Podaj wynik wykonania programu i ciągu poleceń

```
a) 10  LET  x=5
   20  LET  x=x+1
   30  LET  x=3+x
   40  LET  x=-1+x
   50  PRINT  x
```

```
b) a=100
   b=2*a/5
   c=INT( a/b )^INT( b/a*10 )
   d=(a+c) / (b-c)
   PRINT a; b; c; d
```

5.4. Podaj jakie błędy popełniono w zapisie wyrażeń

```
a) x + 2,5 + pcc
b) ((x+y)*(-y)^2))
c) k*  /180
d) a(b+1)
e) LN(x)^2.3
f) (i+j)/(i-SGN(i))
```

5.5. Napisz program obliczania powierzchni trójkąta mając następujące dane:

a) a – podstawa, h – wysokość;
b) a, b, c – trzy boki;
c) a, b – dwa boki, α – kąt między a i b (w stopniach);
d) a – podstawa, α, β – dwa kąty przyległe do podstawy (w stopniach).

5.6. Napisz program obliczania kosztów przejazdu samochodu na trasie o długości d km przy zużyciu k litrów benzyny/100 km i cenie z za 1 litr. Program zaprojektuj w taki sposób, aby łatwo można było dostosować go do różnych typów samochodów.

5.7. Napisz program wyliczania miesięcznych rat spłaty pożyczki p na pr procent i na m miesięcy. Oblicz całkowitą wielkość spłaty. Wzór na obliczenie raty miesięcznej rm ma postać

$$rm = \left[\frac{pr}{1200} \cdot \left(1 + \frac{pr}{1200}\right)^m \cdot p\right] \Big/ \left[\left(1 + \frac{pr}{1200}\right)^m - 1\right]$$

5.8. Napisz program powodujący powstanie nadmiaru. Poprzez kolejne modyfikacje programu spróbuj znaleźć maksymalną dopuszczalną wartość typu liczbowego (czyli taką, której powstanie nie powoduje błędu nadmiaru).

Let's learn BASIC English

expression wyrażenie; **simple** ~ wyrażenie proste
operand argument
addition dodawanie
subtraction odejmowanie
negation negacja
multiplication mnożenie
division dzielenie
power up potęgowanie
overflow nadmiar
underflow niedomiar
function funkcja
roundoff zaokrąglenie
truncation obcięcie
degree stopień (0°..360°)
radian radian (0..2)
ABS (ABSolute) absolutna
FIX (FIXed) ustalona
INT (INTeger) całkowita
SGN (SiGNum) znak
SQR (SQuare Root) pierwiastek
ATN (ArcTaN) arctg
TAN (TANgent) tg

END of lesson

6. Sterowanie obliczeniami

Dotychczas rozważaliśmy programy o strukturze liniowej, których działanie polega na wykonaniu instrukcji w kolejności określonej wzrastającymi numerami linii. Jedynym zakłóceniem w tej sekwencji jest opuszczenie instrukcji komentarza oraz instrukcji danych tworzących osobny strumień danych wewnętrznych.

W opracowywanych algorytmach często jednak pojawiają się alternatywne sekwencje czynności, których wybór jest zależny od dotychczas uzyskanych wyników obliczeń. Również w wielu przypadkach konieczne jest wielokrotne wykonanie tych samych obliczeń dla zmieniających się wartości zmiennych. Zakodowanie tych elementów algorytmu jest możliwe, jeśli skorzysta się w tym celu z instrukcji warunkowych oraz iteracyjnych. Jednocześnie zastosowanie tych instrukcji ma istotny wpływ na strukturę programu, efektywność jego wykonywania oraz czytelność. W takim programie znajdują się już bowiem sekwencje instrukcji, które być może nigdy dla danego zestawu danych nie będą wykonane oraz takie, które będą wykonywane wielokrotnie – nawet tysiące lub więcej razy.

Należy więc o tym pamiętać poznając najprostszy zestaw instrukcji sterujących obliczeniami w programie.

6.1. Wyrażenia relacji

Podjęcie decyzji o wykonaniu kolejnej czynności wymaga sprawdzenia wartości wybranych zmiennych i porównania ich z wartościami stałymi lub między sobą.

Wyrażenie relacji porównuje wartości dwóch wyrażeń typu liczbowego.

wyrażenie_liczbowe operator_relacji wyrażenie_liczbowe

Operator_relacji określa rodzaj porównania.

W języku BASIC wyróżniono następujące operatory relacji:

=	równe
<>	nierówne
>	większe
<	mniejsze
>=	nie mniejsze
<=	nie większe

Proszę zwrócić uwagę na to, że symbolem operatorów nierówne, nie mniejsze i nie większe są dwa znaki. Symbol równości jest identyczny z symbolem przypisania – interpretacja funkcji operatora zależy więc od kontekstu jego umieszczenia w linii. Priorytet operatorów relacji jest identyczny i jest mniejszy od wszystkich operatorów arytmetycznych – tutaj przyjmiemy, że ma wartość równą (3).

Należy też podkreślić to, że w wyrażeniu relacji może wystąpić tylko jeden operator relacji stojący między dwoma wyrażeniami liczbowymi.

Wynik wykonania operacji relacji jest określony przez jedną z dwóch wartości logicznych:

fałsz – jeżeli relacja nie jest spełniona.

prawda – jeżeli relacja jest spełniona.

Pojęcie wartości logicznej jest umowne. W rzeczywistości wartości logiczne są reprezentowane przez wartości liczbowe – *fałsz* jest równe 0, a *prawda* równe –1.

Przykładami wyrażeń relacji są:

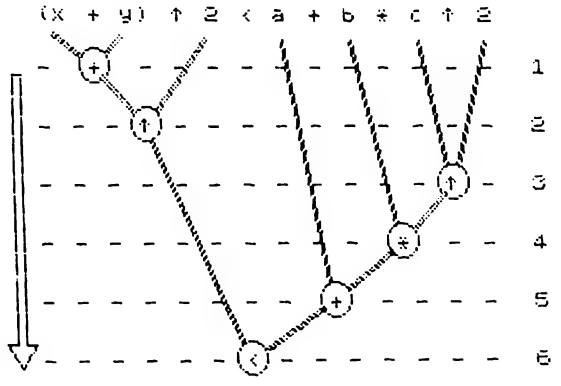
```
i > 5                          LOG(x) > (x-7)^2
x = 0                          b%b - 4%a%c < 0
3 <= 7.85                      SQR(x) <> x^( 1/2 )
```

Przykładowo dla a = 5 i b = 4 wykonanie podanych wyrażeń relacji daje wartości:

a >= b	–1, prawda
a <> b + 1	0, fałsz
a < SIN(b)	0, fałsz
a = b	0, fałsz
a–3 < b	–1, prawda
a <= b + 1	–1, prawda

Rozważmy jeszcze kolejność obliczania pewnego wyrażenia. Kolejność ta jest pokazana na rys. 6.1.

Myślę, że ten przykład jest chyba wystarczający. Muszę jeszcze zwrócić uwagę na konieczność uwzględnienia dokładności obliczeń, jeżeli chcemy od tego uzależniać wybór decyzji co do dalszych czynności.



Rys. 6.1

Przykładowo, dla komputera obliczenie każdego z podanych wyrażeń relacji spowoduje uzyskanie takiego samego wyniku:

1237 => −57 −1, prawda
5E−38 <> 4E−38 −1, prawda

A musisz przyznać Czytelniku, że w drugim wyrażeniu są to dla nas wartości praktycznie identyczne. Dlatego też, dla uzyskania jednoznaczności wyniku relacji w stosunku do założonej dokładności obliczeń, w wątpliwych przypadkach, szczególnie przy sprawdzaniu równości, należy:

a) zamiast relacji równości stosować relacje zawierania w przedziale dopuszczalnego błędu nierówności, a więc dla relacji
wyrażenie 1 = *wyrażenie 2*
lepiej
ABS(*wyrażenie 1* − *wyrażenie 2*) < *epsilon*
gdzie wartość *epsilon* jest dopuszczalnym odstępstwem od równości;

b) stosować obcięcie (lub zaokrąglenie) wartości porównywanych wyrażeń do pożądanej dokładności wartości, a więc dla relacji
wyrażenie 1 = *wyrażenie 2*
lepiej
FIX(*wyrażenie 1*∗100) = FIX(*wyrażenie 2*∗100)
jeżeli założyliśmy dokładność porównania równą 0.01.

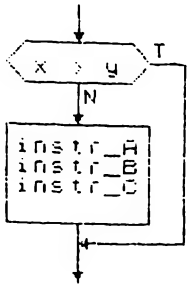
6.2. Instrukcja warunkowa

Zmiana kolejności wykonywania instrukcji może być uzyskana przez analizę wartości wyrażenia relacji w *instrukcji warunkowej.*

IF *wyrażenie_relacji* **THEN** *instrukcja*
IF *wyrażenie_relacji* **THEN** *numer_linii*
IF *wyrażenie_relacji* **GOTO** *numer_linii*

Wykonanie instrukcji powoduje obliczenie wartości *wyrażenia_relacji*. Jeżeli wartość ta jest *prawdą* (czyli jest równa −1), wtedy następuje wykonanie *instrukcji* znajdującej się po słowie kluczowym THEN lub przejście do wykonania instrukcji z linii o podanym *numerze_linii*. W przeciwnym razie następuje wykonanie następnej w kolejności instrukcji.

Zastosowania instrukcji warunkowych zaprezentujemy na kilku prostych przykładach. Dla typowych struktur decyzyjnych algorytmów będziemy poszukiwać odpowiedniego najefektywniejszego zapisu w języku programowania.

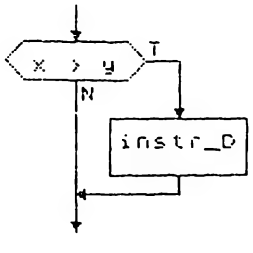


Rys. 6.2

W pierwszym przykładzie, pokazanym na rys. 6.2, spełnienie warunku oznacza opuszczenie wykonania kilku (trzech) instrukcji. Wykorzystamy tutaj dwie pierwsze postacie instrukcji warunkowych

```
10  IF  x>y  THEN  50
20    instr_A
30    instr_B
40    instr_C
50  '...
```

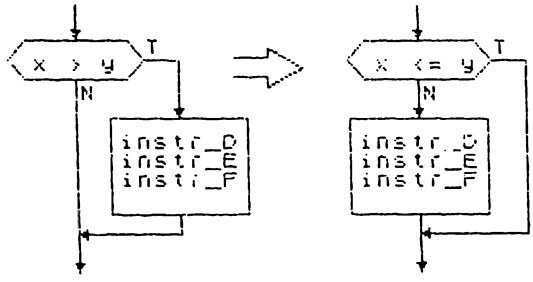
```
10  IF  x>y  GOTO  50
20    instr_A
30    instr_B
40    instr_C
50  '...
```



Rys. 6.3

W kolejnym przykładzie (rys. 6.3) możemy wykorzystać trzecią postać instrukcji warunkowej – jeżeli dla spełnionego warunku należy wykonać tylko jedną instrukcję

```
100  IF  x>y  THEN  instr_D
110  '....
```

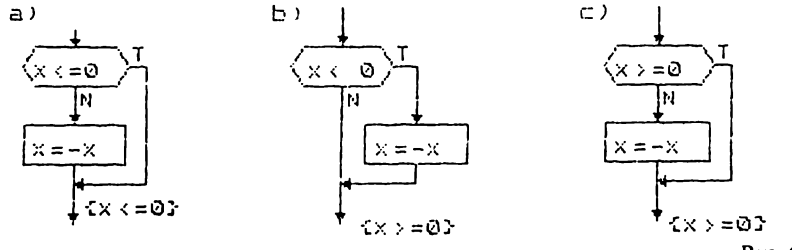


Rys. 6.4

Zwielokrotnienie liczby instrukcji wykonywanych dla warunku spełnionego wymaga przekształcenia sieci działań o postaci z pierwszego rysunku. Otrzymujemy wtedy sieć działań jak na rys. 6.4 i instrukcje

```
100  IF  x<=y  THEN  140
110    instr_D
120    instr_E
130    instr_F
140  '...
```

Podane tutaj zasady stanowią podstawowy element konstruowania poprawnych strukturalnie programów. Prześledźmy więc jeszcze jeden przykład zapisu fragmentu algorytmu pokazanego na rys. 6.5a



Rys. 6.5

W pierwszej wersji po wykonaniu tego fragmentu algorytmu wartość x musi spełniać relację $x \leqslant 0$, a więc należy zapisać instrukcję

```
100  IF  x<=0  THEN  120
110    LET    x = -x
120  REM  x<=0
```

W drugiej wersji przykładu (rys. 6.5b) efektem ma być wartość $x \geqslant 0$. Można zastosować instrukcję

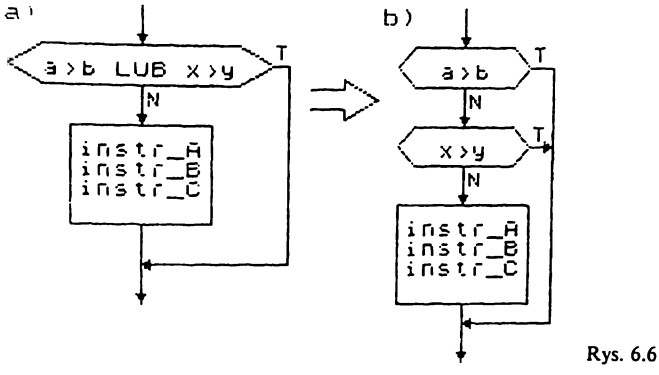
```
100  IF  x<0  THEN  LET  x = -x
120  REM  x>=0
```

lub po przekształceniu sieci (rys. 6.5c)

```
100  IF  x>=0  THEN  120
110    LET  x = -x
120  REM  x>=0
```

Zasada ta, zastosowana tutaj w trywialnej postaci jest dość często wykorzystywana.

Wracamy do naszych typowych struktur decyzyjnego sterowania.



Rys. 6.6

W sieci działań z rys. 6.6a decyzja o opuszczeniu instrukcji jest podejmowana na podstawie alternatywy spełnienia dwóch warunków. Zakodować możemy zaś sieć przekształconą do postaci na rys. 6.6b

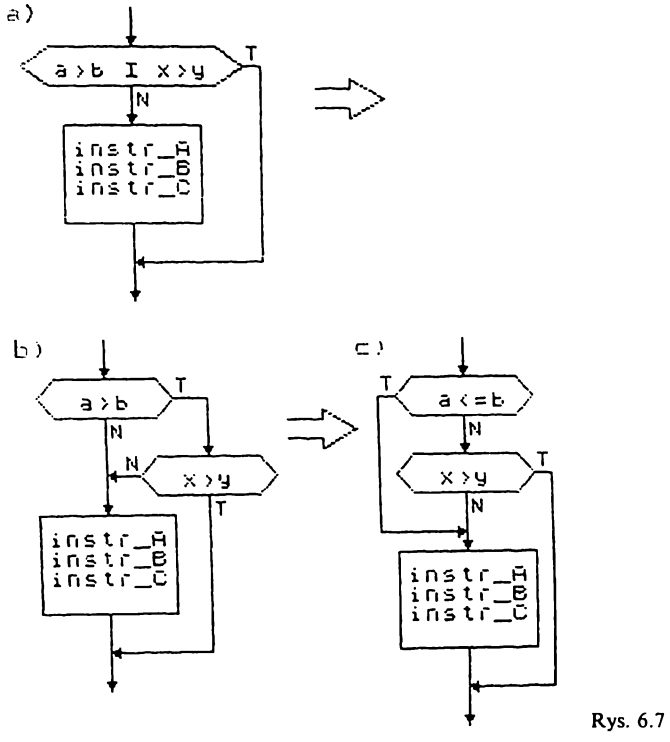
```
50   IF  a>b  THEN  100
60   IF  x>y  THEN  100
70     instr_A
80     instr_B
90     instr_C
100  '...
```

Kolejna sieć działań, z rys. 6.7a, opisuje sytuację z decyzją na podstawie iloczynu logicznego dwóch warunków. Przekształcenie do postaci z rys. 6.7b umożliwia zakodowanie w postaci

```
200  IF  a>b  THEN  IF  x>y  THEN  240
210    instr_A
220    instr_B
230    instr_C
240  '...
```

Instrukcja warunkowa w linii 200; po słowie kluczowym THEN zawiera następną instrukcję warunkową. Drugie rozwiązanie jest oparte na przekształceniu warunków, jak pokazano na rys. 6.7c

```
200  IF  a<=b  THEN  210
205  IF  x> y  THEN  240
210    instr_A
220    instr_B
230    instr_C
240  '...
```



Rys. 6.7

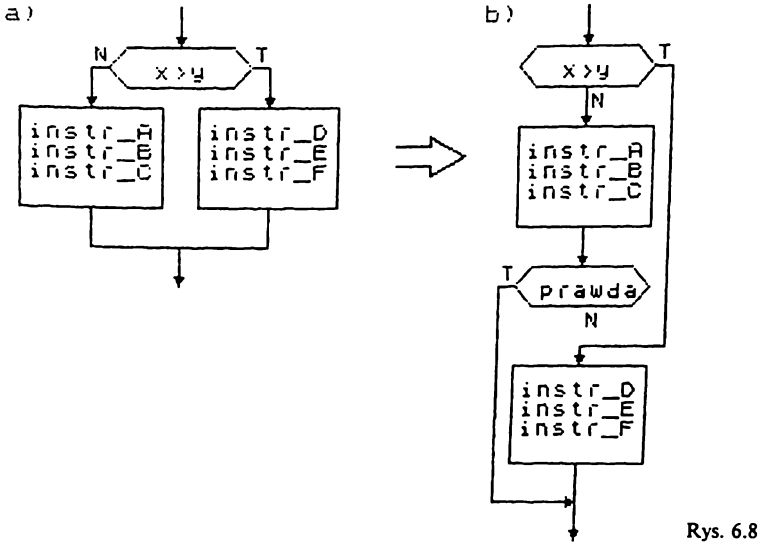
Warto przy okazji wspomnieć, że w przypadku kilku warunków mających jednoczesny wpływ na wybór drogi sterowania, jako pierwszy należy obliczać warunek mający największe prawdopodobieństwo rozstrzygnięcia decyzji.

Dodatkową ilustracją omawianych tutaj zagadnień jest prosty program

```
10  INPUT  x
20  IF  x<0  THEN PRINT "jest ujemne  "
30  IF  x=0  THEN PRINT "jest zerem   "
40  IF  x>0  THEN PRINT "jest dodatnie"
50  REM  W przykladzie tym wykona sie
51  '     zawsze tylko jedna z instrukcji
52  '     wyprowadzania z linii 20,30,40
```

Sieć działań, pokazana na rys. 6.8a, prezentuje nieco bardziej skomplikowaną strukturę sterowania. Zakodowanie tego przykładu, za pomocą aktualnie zaprezentowanych instrukcji wymaga przekształcenia sieci do postaci przedstawionej na rys. 6.8b. Zapis fragmentu programu jest następujący

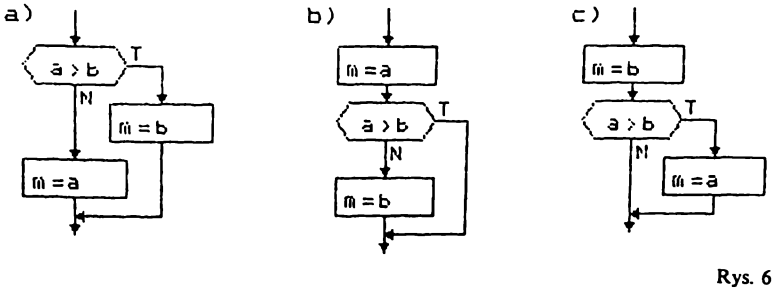
```
100  IF  x>y  THEN  125
105    instr_A
110    instr_B
115    instr_C
120    IF  -1  THEN  140
125  instr_D
130  instr_E
135  instr_F
140  '...
```



Rys. 6.8

W instrukcji z linii 120 zawsze będzie wykonane przejście do linii 140.

Dla odprężenia, przeanalizujmy rozwiązanie następującego, prostego problemu. Spośród dwóch wczytanych wartości należy wyprowadzić wartość



Rys. 6.9

większą. Sieć działań algorytmu jest pokazana na rys. 6.9a. Bezpośrednie zakodowanie nie jest jednak takie proste. Łatwiejsze do zakodowania jest rozwiązanie z rys. 6.9b lub 6.9c. Programy mają wtedy postać

```
a)  10  INPUT  a,b
    20  LET  m=a
    30  IF  a>b  THEN  40
    35    LET  m=b
    40  PRINT  m
    50  END

b)  10  INPUT  a,b
    20  LET  m=b
    30  IF  a>b  THEN  LET  m=a
    35     'brak instrukcji
    40  PRINT  m
    50  END
```

Czasem opłaca się dokonać operacji na wyrost, a potem ewentualnie zmodyfikować wartość za pomocą nowej operacji.

Przykład 6.1. „Porządkowanie trzech wartości"

Opis problemu

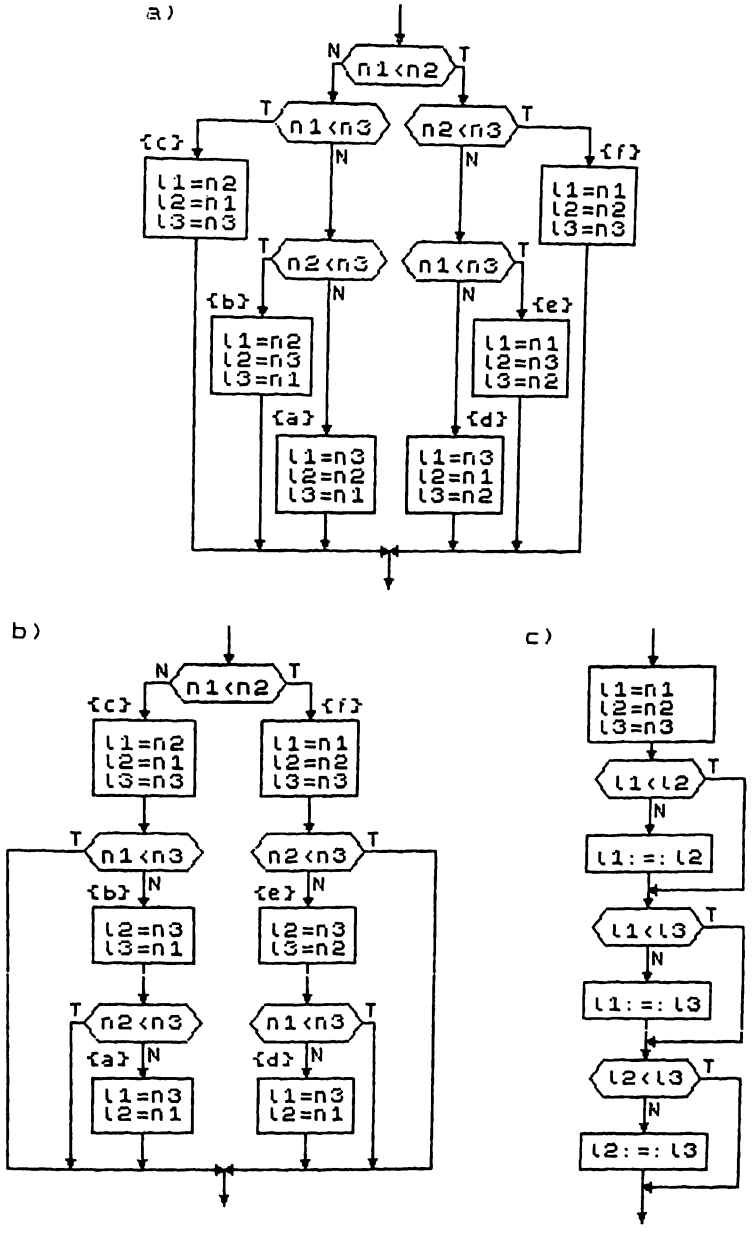
Dla trzech wczytanych z klawiatury wartości, chcemy wyprowadzić je w kolejności wzrastającej.

Metoda rozwiązania

Zadanie jest trywialne, ale tutaj będziemy poszukiwać najlepszej metody rozwiązania. Pierwszy algorytm nasuwa się od razu. Parami sprawdzamy wprowadzone wartości, umieszczając je w zmiennych *l1*, *l2*, *l3* zawierających w rezultacie wartości uporządkowane wzrastająco. Sieć działań tego rozwiązania przedstawia rys. 6.10a. Analizując ten algorytm widzimy, że można niektóre instrukcje przypisania poprzenosić przed badanie relacji oraz niektóre połączyć dla dwóch alternatyw. Otrzymamy więc sieć działań jak na rys. 6.10b. W sieci tej można jeszcze poprzenosić niektóre obliczenia przed badanie relacji otrzymując tylko jedną alternatywę z dwoma sekwencjami obliczeń. Nasze działania zmierzają we właściwym kierunku – poszukajmy więc jeszcze lepszego. Będziemy lokalnie sprawdzać relacje sąsiednich wartości, zamieniając je ewentualnie miejscami. Rozwiązanie to przedstawiono na rys. 6.10c.

Teksty programów

W celu porównania zapiszemy teksty trzech rozwiązań: dla sieci działań z rys. 6.10a, 6.10b (zmodyfikowanego) i 6.10c. (Dla oszczędności miejsca zamieszczono tylko podstawowe części programu z ograniczoną liczbą komentarzy.)


a)
n1<n2
n1<n3
n2<n3
n2<n3
n1<n3
{c}
l1=n2 l2=n1 l3=n3
{f}
l1=n1 l2=n2 l3=n3
{b}
l1=n2 l2=n3 l3=n1
{e}
l1=n1 l2=n3 l3=n2
{a}
l1=n3 l2=n2 l3=n1
{d}
l1=n3 l2=n1 l3=n2
b)
n1<n2
{c}
l1=n2 l2=n1 l3=n3
{f}
l1=n1 l2=n2 l3=n3
n1<n3
n2<n3
{b}
l2=n3 l3=n1
{e}
l2=n3 l3=n2
n2<n3
n1<n3
{a}
l1=n3 l2=n1
{d}
l1=n3 l2=n1
c)
l1=n1 l2=n2 l3=n3
l1<l2
l1:=:l2
l1<l3
l1:=:l3
l2<l3
l2:=:l3

Rys. 6.10

```
5    REM  =A=  (wg.rys.6.10a)
10   '    ----------------------------------------
20   INPUT  n1,n2,n3
30   '
100  REM   Poczatek porzadkowania
105  '
110  IF  n1<n2  THEN  260
120    IF  n1<n3  THEN  220
130      IF  n2<n3  THEN  180
140        LET  l1=n3              'a
150        LET  l2=n2
160        LET  l3=n1
170        GOTO   400
180      LET  l1=n2                'b
190      LET  l2=n3
200      LET  l3=n1
210      GOTO   400
220    LET  l1=n2                  'c
230    LET  l2=n1
240    LET  l3=n3
250    GOTO   400
260  IF  n2<n3  THEN  360
270    IF  n1<n3  THEN  320
280      LET  l1=n3                'd
290      LET  l2=n1
300      LET  l3=n2
310      GOTO   400
320    LET  l1=n1                  'e
330    LET  l2=n3
340    LET  l3=n2
350    GOTO   400
360  LET  l1=n1                    'f
370  LET  l2=n2
380  LET  l3=n3
390  GOTO   400
395  '
400  REM  Koniec porzadkowania
500  PRINT  l1;l2;l3
900  END
```

```
5    REM  =B=  (wg.rys.6.10b)
10   '    ----------------------------------------
20   INPUT  n1,n2,n3
30   '
100  REM   Poczatek porzadkowania
105  '
110  IF  n1<n2  THEN  260
120    LET  l1=n2                  'c
130    LET  l2=n1
140    LET  l3=n3
150    IF  n1<n3  THEN  400
160      LET  l2=n3                'b
170      LET  l3=n1
180      IF  n2<n3  THEN  400
190        LET  l1=n3              'c
200        LET  l2=n2
210        GOTO   400
250  '
260  LET  l1=n1                    'f
270  LET  l2=n2
280  LET  l3=n3
290  IF  n2<n3  THEN  400
300    LET  l2=n3                  'e
310    LET  l3=n2
320    IF  n1<n3  THEN  400
330      LET  l1=n3                'd
340      LET  l2=n1
350      GOTO   400
395  '
400  REM  Koniec porzadkowania
500  PRINT  l1;l2;l3
900  END
```

```
5    REM  =C=  (wg.rys.6.10c)
10   '    ----------------------------------------
20   INPUT  n1,n2,n3
30   '
100  REM   Poczatek porzadkowania
105  '
110  LET  l1=n1
120  LET  l2=n2
130  LET  l3=n3
140  IF  l1<l2  THEN  180
150    LET  p=l1        'mozna krocej     l1=n2
160    LET l1=l2        'mniej elegancko  l2=n1
170    LET l2=p
180  IF  l1<l3  THEN  220
190    LET  p=l1        'rownie: krocej   l3=l1
200    LET l1=l3        'mniej elegancko  l1=n3
210    LET l3=p
220  IF  l2<l3  THEN  400
230    LET   p=l2       'tu juz nie mozna
240    LET  l2=l3       'inaczej
250    LET  l3=p
260    GOTO   400
395
400  REM  Koniec porzadkowania
500  PRINT  l1;l2;l3
900  END
```

Komentarz do rozwiązania

Spróbujmy dokonać oceny tych trzech programów. Dokonamy oceny postaci zapisu tekstu programu oraz efektywności jego działania. Kryterium porównawczym postaci zapisu będzie liczba instrukcji przypisania i warunkowych, jaka została użyta w programie. Kryterium porównawczym działania programu będzie średnia liczba wykonanych instrukcji przypisania i średnia liczba wykonanych instrukcji warunkowych obliczona dla wszystkich możliwych początkowych uporządkowań danych. Pomijając możliwość wystąpienia równości w wartościach danych, mamy sześć różnych uporządkowań początkowych.

Wyniki tej analizy są zawarte w tablicy.

PROGRAM	A	B	C
Liczba użytych instrukcji			
– przypisania	18	14	12
– warunkowych	5	5	3
Średnia liczba wykonanych instrukcji			
– przypisania	3	5	7.5
– warunkowych	2.66	2.66	3

Koniec przykładu.

Na podstawie uzyskanych wyników dla, bądź co bądź, bardzo prostego problemu możemy wysnuć jeden z najważniejszych wniosków charakteryzujących proces programowania.

Efektywność programu. Praktycznie dla każdego problemu istnieją algorytmy, których zakodowanie wymaga mniejszej lub większej liczby instrukcji. Zmniejszenie liczby instrukcji koniecznych do zapisania programu pociąga za sobą z reguły zwiększenie czasu jego działania (oraz odwrotnie).

Do tego zagadnienia będziemy jeszcze powracać. Należy zaznaczyć, że podane tutaj rozwiązania nie są najoptymalniejsze. Poznając kolejne rozszerzenia języka BASIC będziemy mogli jeszcze poprawić ten program.

6.3. Cykliczne wykonywanie instrukcji

W dotychczasowych strukturach sterowania obliczeniami nie było powrotu do już raz wykonanych instrukcji.

Rozważmy więc prosty program znajdowania wartości średniej dla 10 wartości wczytywanych z klawiatury. W drugiej kolumnie zapisałem inną wersję tego programu i to na tyle uniwersalną, że z jego pomocą można liczyć wartości średnie dla dowolnej liczby wartości (oczywiście jeżeli po drodze nie wystąpi nadmiar). O wyższości drugiego rozwiązania nie trzeba chyba nikogo przekonywać.

```
100  LET s=0
110  INPUT x
120  LET s=s+x
130  INPUT x
140  LET s=s+x
150  INPUT x
160  LET s=s+x
170  INPUT x
180  LET s=s+x
190  INPUT x
200  LET s=s+x
210  INPUT x
220  LET s=s+x
230  INPUT x
240  LET s=s+x
250  INPUT x
260  LET s=s+x
270  INPUT x
280  LET s=s+x
290  INPUT x
300  LET s=s+x
310  PRINT "suma        :";s
320  PRINT "srednia     :";s/10
330  END
```

```
100  LET s=0
105  INPUT "Liczba elementow"; n
110  LET n1=n
210    INPUT x
220    LET s=s+x
230    LET n=n-1
240    IF  n>0  THEN  210
310  PRINT "suma        :";s
320  PRINT "srednia     :";s/n1
330  END
```

Cyklem w programie nazywamy sekwencję instrukcji, które mogą być wielokrotnie wykonywane. Wyróżniamy cykle *iteracyjne* i *wyliczane.*

W podanym przykładzie cykl tworzą instrukcje z linii 10, 20, 25, a instrukcja z linii 30 jest instrukcją sterowania wykonywaniem kolejnych cykli.

Zanim uogólnimy pojęcie cyklu omówimy *instrukcję przejścia* (czasem zwaną też *instrukcją skoku*).

GOTO *numer_linii*

Wykonanie instrukcji powoduje bezwarunkowe przejście do linii o podanym *numerze_linii.* Przyjmiemy, że jest to skrócony zapis instrukcji

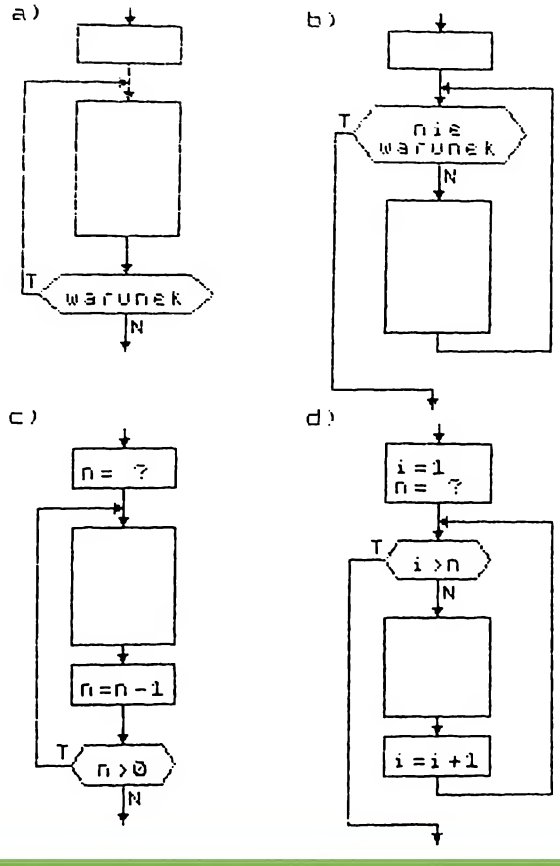
IF −1 GOTO *numer_linii*

Na początek rozważmy przykład

```
10  REM  POWTORZ
20    PRINT  "Entliczek-Pentliczek"
30    GOTO  10
40  REM  WROC ZAWSZE
RUN
Entliczek-Pentliczek
Entliczek-Pentliczek
Entliczek-Pentliczek
Entliczek-Pentliczek
Entliczek-Pentliczek
Entliczek-Pentliczek
Entliczek-Pentliczek
        [CTRL-C]
Break in line 20
Ok
```

Jest to przykład cyklu nieskończonego, którego wykonywanie musi być przerwane z zewnątrz.

Nieeleganckie przerywanie wykonywania instrukcji wewnętrznych cyklu może być zastąpione programowym sprawdzaniem wartości warunku powtarzania decydującym o wykonaniu kolejnego cyklu. Warunek ten może być umieszczony na początku lub na końcu sekwencji instrukcji zawartych w cyklu. W pierwszym przypadku decyzja zapada przed każdym kolejnym wykonywaniem cyklu, w drugim – po jego wykonaniu. W organizacji instrukcji cyklu wyróżniamy też instrukcje ustalania wartości początkowych dla zmiennych biorących udział w wyznaczaniu wartości warunku powtarzania (rys. 6.11).



Rys. 6.11

Wykonywanie cyklu. Zaniechanie wykonywania cyklu może nastąpić na podstawie dwóch kryteriów.

1. *Cykl iteracyjny.* Warunek kończący cykl jest wyrażeniem relacji zawierającym też zmienną lub zmienne o wartościach wyznaczonych przez instrukcje zawarte w cyklu. Każde kolejne obliczenie przybliża uzyskanie wartości właściwej. W cyklach iteracyjnych nie można często wcześniej określić, ile cykli będzie wykonanych.
2. *Cykl wyliczany.* Warunek kończący cykl jest określony wartością zmiennej zliczającej liczbę wykonywanych cykli. Zmienna ta zwana jest *zmienną sterującą* i musi być zainicjowana *wartością początkową*, po czym jej wartość jest sprawdzana z założoną *wartością końcową* – określającą z góry liczbę wykonań cykli.

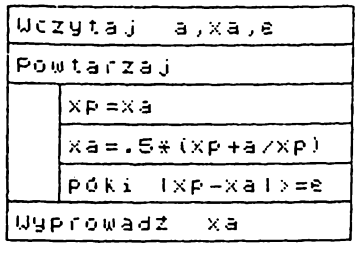
Dla sieci z rys. 6.11a zapis cyklu iteracyjnego ma postać:

```
140  '...
150  REM  Powtarzaj cykl
160    instr_A
170    instr_B
180    instr_C
190    IF  warunek  THEN  150
200  REM  dopoki warunek jest spelniony
210  '...
```

Dla schematu z rys. 6.11b zapis cyklu iteracyjnego ma z kolei postać:

```
140  REM  Ustal stan poczatkowy zmiennych dla warunku
150  REM  Powtarzaj cykl poki warunek jest spelniony
155  IF  nie warunek  THEN  200
160    instr_A
170    instr_B
180    instr_C
190    GOTO  150
199  REM  wroc na poczatek
200  '...
```

Tutaj w celu zachowania jednolitości spełnienia warunku dla wykonania kolejnego cyklu zastosowałem w instrukcji z linii 155 zaprzeczenie warunku. Czytelnikowi nie powinno sprawić kłopotu właściwe jego zapisanie.
Prostym przykładem zastosowanych tutaj konstrukcji jest program iteracyjnego obliczania wartości pierwiastka zapisany w diagramie na rys. 6.12



Rys. 6.12

```
1   REM  Program iteracyjnego obliczania
2   '    pierwiastka kwadratowego z danej
3   '    liczby wedlug wzoru Newtona:
4   '
5   '    x2=0.5*(x1+a/x1)
6   '
7   '    z zadana dokladnoscia e-epsilon
8   '
10 PRINT  "Podaj:"
15 INPUT  "  liczbe      :"; a
20 INPUT  "  wart.pocz   :"; xa
25 INPUT  "  dokladnosc  :"; e
30 REM  Poczatek obliczen iteracyjnych
35   xp=xa
40   xa=0.5 * ( xp + a/xp )
45   IF  ABS(xp-xa) >= e  THEN  30
50 REM  dopoki roznica kolejno uzyskanych
55 '    wartosci jest nie mniejsza od
60 '    epsilon
65 PRINT "Wartosc pieriastka :"; xa
70 END
```

Obie zmienne *xp* i *xa* przechowują odpowiednio poprzednią i aktualną obliczoną wartość. Przy wykorzystywaniu iteracji należy pamiętać o tym, aby formuła iterowanego wyznaczania wartości dawała proces zbieżny. Zagadnienie to jest częścią matematycznej analizy efektywności zastosowanego algorytmu.

Dla sieci z rys. 6.11c zapis cyklu wyliczanego ma postać

```
140  LET  n= liczba_cykli
150  REM  Powtarzaj cykl
160    instr_A
170    instr_B
180    instr_C
190    LET n = n-1
195    IF  n>0  THEN  150
199  REM  dopoki  n>0
200  '...
```

Dla sieci z rys. 6.11d zapis cyklu wyliczanego ma postać

```
140  LET   i = 1
145  LET   n = liczba_cykli
150  REM  Powtarzaj cykl poki  i<=n
155  IF  i>n  THEN  200
160    instr_A
170    instr_B
180    inst_C
190    LET  i = i+1
195    GOTO  150
199  REM  wroc na poczatek
200  '...
```

Ilustracją pierwszego rozwiązania cyklu wyliczanego jest program zamieszczony na początku tego podrozdziału. Rozwiązanie to może być zastosowane, jeżeli z góry znamy liczbę koniecznych do wykonania cykli oraz zakładamy, że instrukcje cyklu mają się wykonać przynajmniej jeden raz. Ilustracja drugiego rozwiązania cyklu wyliczanego zostanie podana w następnym podrozdziale przy omawianiu nowej instrukcji języka, upraszczającej zapis tego schematu obliczeń.

6.4. Instrukcje cyklu wyliczanego

W zapisie cyklu wyliczanego skonstruowanego na podstawie schematu z rys. 6.11d jest kilka instrukcji organizujących kontrolę sterowania cyklem. Dla ułatwienia zapisu organizacji cyklu wyliczonego w języku istnieje *instrukcja cyklu wyliczonego* podzielona na dwie instrukcje określające początek i koniec sekwencji instrukcji zawartej w cyklu.

FOR *zm_sterująca*=*wart_początk* **TO** *wart_końc* **STEP** *wart_kroku*

FOR *zm_sterująca*=*wart_początk* **TO** *wart_końc*
NEXT *zm_sterująca*

Zmienna_sterująca musi być prostą zmienną typu liczbowego i przyjmuje ona kolejno wartości począwszy od *wartości_początkowej* z inkrementem (lub dekrementem) określonym *wartością_kroku* aż do przekroczenia *wartości_końcowej*. Wszystkie trzy wartości, będące parametrami cyklu wyliczanego, są wyrażeniami arytmetycznymi. Brak ostatniej części instrukcji – określonej słowem kluczowym STEP – oznacza przyjęcie wartości kroku równej 1.

Dla *instrukcji początku cyklu* (FOR...TO) parą jest *instrukcja jego końca* NEXT powiązana przez tą samą *zmienną_sterującą*. Zmienna ta ma identyczne własności jak pozostałe zmienne programu, przy ograniczeniu działania do zakresu instrukcji cyklu.

Przy opisie działania tych instrukcji skorzystam jeszcze z opisów prezentowanych w poprzednim punkcie. Dla uproszczenia przyjmijmy przykładowy zapis instrukcji cyklu

```
100 FOR i=wartosc_poczatkowa TO wartosc_koncowa STEP wartosc_kroku
      ciag_instrukcji_cyklu
200 NEXT i                                        i - zmienna_sterujaca
```

Poniżej podano zapis z wykorzystaniem poprzednio wprowadzanych instrukcji. Zmienne *GRANICA*, *KROK* są zmiennymi dostępnymi jedynie translatorowi języka BASIC.

```
        GRANICA = wartosc_koncowa
        KROK    = wartosc_kroku
        i       = wartosc_poczatkowa - KROK
100     i       = i + KROK
        IF  SGN( KROK ) * (i - GRANICA) > 0  THEN  200
            ciag_instrukcji_cyklu
        GOTO  100
200     REM   Koniec_instrukcji_cyklu
```

Wykonywanie instrukcji cyklu wyliczanego.

1. Jako pierwsze są obliczone wartości wyrażeń arytmetycznych dla parametrów: *wartość_końcowa* i *wartość_kroku*. Wartości te są wyznaczone tylko raz i dalej są wartościami stałymi podczas wykonywania instrukcji cyklu.
2. Zmiennej sterującej jest następnie przypisywana wartość wyrażenia *wartości_początkowej*. Opóźnienie wyznaczenia wartości zmiennej sterującej zapobiega ewentualnemu wpływowi wartości początkowej na obliczenie wartości końcowej lub kroku.

3. Instrukcje zawarte w cyklu wykonuje się tyle razy, ile razy wartość zmiennej sterującej, modyfikowana w każdym cyklu wartością kroku, nie przekracza wartości końcowej. Modyfikacja wartości zmiennej sterującej może być dodatnia lub ujemna — wtedy odpowiednio wartość końcowa musi być większa lub mniejsza od wartości początkowej.
4. Przekroczenie wartości zmiennej sterującej w stosunku do wartości końcowej powoduje przejście do wykonywania instrukcji następnej po instrukcji końca cyklu. Możliwe jest, że dla danej wartości początkowej, końcowej i kroku instrukcje wewnątrz cyklu nie będą wykonane ani razu.
5. Zmienna sterująca może być wykorzystywana wewnątrz cyklu tylko jako wartość stała (nie może więc być zmieniona). Po zakończeniu wykonywania instrukcji cyklu, wartość zmiennej sterującej może być nieokreślona.
6. Instrukcje wewnątrz sekwencji cyklu mogą być wykonywane jedynie wtedy, kiedy wykonano instrukcję początku cyklu. Inaczej mówiąc nie wolno dokonywać bezpośredniego przejścia do instrukcji zawartych w cyklu. Dopuszczalne jest za to wykonanie instrukcji bezwarunkowego lub warunkowego przejścia na zewnątrz cyklu.

Uwaga: W niektórych wersjach implementacji języka BASIC, wyrażenie wartości początkowej jest obliczane jako pierwsze, a wtedy w obliczaniu wartości końcowych i kroku uwzględnia się ewentualnie nową wartość zmiennej sterującej. Również możliwe jest zmienianie wartości zmiennej sterującej przez instrukcje zawarte w cyklu.

Przeanalizujmy następujące przykłady funkcjonowania instrukcji cyklu — od najprostszych do bardziej skomplikowanych.

```
5   REM  Wersja najprostsza
6   '
10  FOR i=1 TO 6
20    PRINT  i;
30  NEXT i
RUN
 1  2  3  4  5  6
Ok
```

```
5   REM  Krok dodatni - rozny od zera
6   '
10  FOR j=-3 TO 7 STEP 2
20    PRINT  j;
30  NEXT j
RUN
-3 -1  1  3  5  7
Ok
```

```
5   REM  Krok ujemny
6   '
10  FOR k=40 TO 10 STEP -8
20    PRINT  k;
30  NEXT k
RUN
 40  32  24  16
Ok
```

```
5   REM  Zliczanie w wartosciach
6   '    niecalkowitych
10  FOR x=0.2 TO 1.2 STEP 0.4
20    PRINT  x;
30  NEXT x
40  PRINT
50  PRINT  "Poza cyklem ="; x
RUN
 0.2  0.6  1
Poza cyklem = 1.4
Ok
```

```
5   REM  Wyrazenia wartosci poczatkowej,
6   '    koncowej i kroku
10  LET  l=15
20  FOR i=l-5 TO l+20 STEP l/3
30    PRINT  i;
40  NEXT i
RUN
 10  15  20  25  30  35
Ok
```

```
5   REM  To samo, ale z wplywem
6   '    wartosci zmiennej sterujacej
10  LET i=15
20  FOR i=i-15 TO i+20 STEP INT(i/3)
30    PRINT  i;
40  NEXT i
RUN
 10  15  20  25  30  35
Ok
```

```
REM  We wczesniejszych wersjach systemu BASIC
Ok
REM  otrzymamy inny wynik !
Ok
REM  10  13  16  19  21  24  27  30
Ok
NEW
Ok
```

```
5   REM  Pomijanie wykonywania instrukcji
6   '    w cyklu
10  LET k=10
20  FOR j=k$3 TO k/10
21                'k$3 > k/10  - instrukcja
22                'z linii 30 nie wykona sie.
30    PRINT  j;
40  NEXT j
50  PRINT
60  PRINT  "Poza cyklem ="; j
RUN
Poza cyklem = 30
Ok


5   REM  Pomijanie wykonywania instrukcji
6   '    cyklu dla niektorych wartosci
7   '    zmiennej sterujacej
10  FOR k=1 TO 20 STEP 2
20    IF INT(k/3)$3=k THEN 40
30      PRINT k;
40  NEXT k
RUN
 1  5  7  11  13  17  19
Ok
NEW
Ok


5   REM  Wczesniejsze zakonczenie wykony-
6   '    wania instrukcji cyklu
10  INPUT  m
20  FOR j=1 TO 50 STEP 2
30    IF j>=m THEN 60
40      PRINT  j;
50  NEXT j
60  PRINT
70  PRINT  "Poza cyklem ="; j
RUN
? 9
 1  3  5  7
Poza cyklem = 9
Ok


5   REM  Kumulowanie bledu sumowania war-
6   '    tosci zmiennej sterujacej
10  LET lc=0
20  FOR x=-60 TO 60 STEP 0.01
30    LET lc=lc+1
40  NEXT x
50  REM  Oczekiwana wartosc lc=12001
60  PRINT "Liczba cykli ="; lc
70  PRINT "Wartosc  x   ="; x
RUN
Liczba cykli = 12000
Wartosc  x   = 60.00255
Ok
REM  Wynik ten otrzymano na zaprzy-
Ok
REM  jaznionym komputerze
Ok
REM  Sprawdz na swoim !?!
Ok
REM  UWAGA !
Ok
REM  Czas wykonywania tego programu
Ok
REM       jest zauwazalny
Ok
```

```
5   REM Poprzeddni przyklad ze zmiana
6    '   wartosci kroku
20  FOR x=-60 TO 60 STEP 0.1
RUN
Liczba cykli = 1201
Wartosc  x   = 60.0999973
Ok
REM  Liczba cykli jest Ok !
Ok
```

```
5   REM  Prawidlowe rozwiazanie takiej
6    '     instrukcji cyklu
10  LET lc=0
20  FOR i=-60000 TO 60000
25    LET  x=i * 0.01
30    LET lc=lc+1
40  NEXT i
50  REM  Oczekiwana wartosc lc=12001
60  PRINT "Liczba cykli ="; lc
70  PRINT "Wartosc  x   ="; x
RUN
Liczba cykli = 12001
Wartosc  x   = 60
Ok
```

Przy okazji – nie należy korzystać z żadnych efektów ubocznych działania instrukcji cyklu, a więc:

a) dokonywać wewnątrz cyklu zmiany wartości zmiennej sterującej,

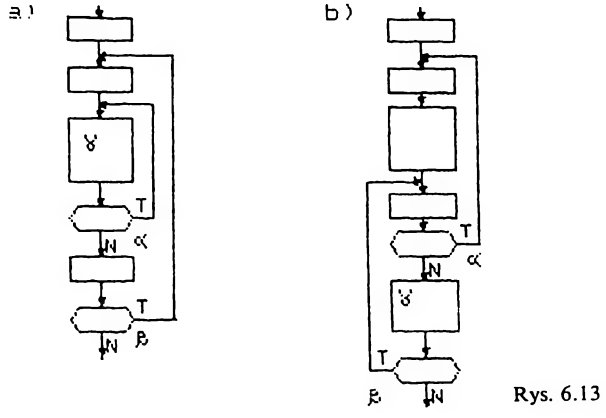
b) wykorzystywać ostatnią wartość zmiennej sterującej po zakończeniu cyklu – z wartości tej możemy korzystać, jeżeli opuściliśmy cykl za pomocą instrukcji warunkowej lub przejścia.

```
10  FOR i=1 TO 20
20    LET i=i*2
30  NEXT i
40  LET x=i*5
```

6.5. Zagnieżdżone instrukcje cyklu

Cechą charakterystyczną wielu struktur programów jest wielokrotne cykliczne wykonywanie tych samych sekwencji instrukcji. Często też jest konieczne powtarzanie wykonywania większych fragmentów programu zawierających już cykle.

Zagnieżdżone instrukcje cyklu. Są to instrukcje zawarte w sekwencjach instrukcji wykonywanych w cyklu. Zasadą konstruowania tego typu instrukcji jest to, aby cykl zagnieżdżony w całości był zawarty w cyklu zewnętrznym.



Rys. 6.13

Podane wymaganie ilustruje rys. 6.13.

Prawidłowo zagnieżdżone instrukcje cyklu (rys. 6.13a) tworzą uporządkowany schemat sterowania. Przeanalizujmy wykonywanie się programu opisanego tym schematem. Załóżmy, że po fazie inicjacji cyklu warunek powtarzalności cyklu zewnętrznego będzie spełniony β razy, warunek zaś powtarzalności cyklu wewnętrznego będzie spełniony α razy. W takim przypadku obliczenia γ wewnątrz cyklu zagnieżdżonego będą wykonane $\gamma = \alpha * \beta$ razy.

Z kolei łańcuchowe powiązanie instrukcji cyklu przedstawia rys. 6.13b. W tym przypadku analiza wykonywania się tego programu jest utrudniona. Instrukcje oznaczone jako γ będą wykonywane po niespełnieniu warunku powtarzalności pierwszego cyklu, ale wtedy zostaje domknięty cykl drugi i powstaje pytanie o stan warunku powtarzalności pierwszego cyklu. Można tutaj jedynie zakładać, że obliczenia γ mają wpływ na stan tego warunku i ewentualne powtórne wykonywanie cyklu α.

Szczególne zastosowanie w zagnieżdżaniu instrukcji cyklu mają instrukcje cyklu wyliczonego.

Warunki zagnieżdżania instrukcji cyklu wyliczonego.

1. Każda kolejna zagnieżdżona instrukcja musi mieć odrębną zmienną sterującą.
2. Wszystkie zagnieżdżone instrukcje muszą być całkowicie zawarte w sekwencji instrukcji cykli dla nich zewnętrznych.
3. Dopuszczalne jest dowolne przejście z instrukcji wewnątrz cyklu na zewnątrz na dowolny poziom.

Dla przykładu porównajmy dwa fragmenty programów.

```
100  FOR i=0 TO 5
110    FOR j=1 TO 4
120      PRINT  i;j;" , ";
130    NEXT j
140    PRINT
150  NEXT i
RUN
 0  1  ,  0  2  ,  0  3  ,  0  4  ,
 1  1  ,  1  2  ,  1  3  ,  1  4  ,
 2  1  ,  2  2  ,  2  3  ,  2  4  ,
 3  1  ,  3  2  ,  3  3  ,  3  4  ,
 4  1  ,  4  2  ,  4  3  ,  4  4  ,
 5  1  ,  5  2  ,  5  3  ,  5  4  ,
Ok
```

```
100  FOR i=0 TO 5
110    FOR j=1 TO 4
120      PRINT  i;j;" , ";
130    NEXT i
140    PRINT
150  NEXT j
```

Pierwszy fragment jest zapisem poprawnym. Wykonanie tego programu spowoduje wyprowadzenie kolejno par liczb. Instrukcja wyprowadzania wartości z linii numer 120 zostanie wykonana $5 \times 4 = 20$ razy.

Drugi fragment programu jest niepoprawny. Instrukcja z linii 130 domyka cykl zewnętrzny – powodując zerwanie wykonywania cyklu wewnętrznego. Zachowanie tego programu będzie nieokreślone i w większości realizacji zostanie wyprowadzony komunikat o błędzie.

Przeanalizujmy jeszcze działanie kilku konstrukcji instrukcji zagnieżdżonych.

```
5    REM  Rozne wartosci kroku
10   REM
20   FOR i=0 TO 10 STEP 3
30     FOR j=-5 TO 0 STEP 2
40       PRINT i;j
50     NEXT j
60   NEXT i
RUN
 0 -5
 0 -3
 0 -1
 3 -5
 3 -3
 3 -1
 6 -5
 6 -3
 6 -1
 9 -5
 9 -3
 9 -1
Ok
```

```
5    REM  Wartosc koncowa uzalezniona
10   REM  od wartosci zmiennej
15   REM  sterujacej
20   FOR n=1 TO 4
30     FOR m=0 TO n
40       PRINT  n;m
50     NEXT m
60   NEXT n
RUN
 1  0
 1  1
 2  0
 2  1
 2  2
 3  0
 3  1
 3  2
 3  3
 4  0
 4  1
 4  2
 4  3
 4  4
Ok
```

```
R5    REM  Wartosc poczatkowa cyklu
10   REM  zalezna od zmiennej
15   REM  sterujacej
20   FOR i=1 TO 5
30     FOR j=i TO 4
40       PRINT i;j
50     NEXT j
60   NEXT i
RUN
 1  1
 1  2
 1  3
 1  4
 2  2
 2  3
 2  4
 3  3
 3  4
 4  4
Ok
' Zauwazmy, ze dla i=5 cykl
Ok
'wewnetrzny nie wykonal sie
Ok
```

```
5    REM  Wartosc koncowa i krok
10   REM  zalezne od zmiennej
15   REM  sterujacej
20   FOR a=1 TO 4
30     FOR b=1 TO a*3 STEP a
40       PRINT a;b
50     NEXT b
60   NEXT a
RUN
 1  1
 1  2
 1  3
 2  1
 2  3
 2  5
 3  1
 3  4
 3  7
 4  1
 4  5
 4  9
Ok
```

Styl programowania

Zakaz wykorzystywania instrukcji GOTO. Instrukcji przejścia bezwarunkowego nie wolno używać w programowaniu. To ostre stwierdzenie należy w przypadku języka BASIC potraktować trochę na wyrost. W ogólnym przypadku, w najnowszych językach wysokiego poziomu z dobrze określonymi instrukcjami sterowania wykonywaniem programu – nie ma potrzeby używać tej instrukcji. Co więcej, brak tej instrukcji umożliwia uzyskać elegancki, dobrze zbudowany, strukturalny program. W języku BASIC na razie nie możemy się pozbyć tej instrukcji ale w miarę poznawania kolejnych coraz lepszych instrukcji z rozszerzeń języka będziemy mogli zmniejszyć częstotliwość jej używania. W każdym razie niech każde użycie instrukcji przejścia wzbudza w nas niepokój, a na pewno nie powinniśmy nigdy otrzymać programu o strukturze typu

```
10    GOTO   100
20    PRINT  "Y";
30    GOTO   170
40    PRINT  "!"
50    GOTO   140
60    PRINT  "R";
70    GOTO   150
80    PRINT  "O";
90    GOTO   40
100   PRINT  "B";
110   GOTO   60
120   PRINT  "K";
130   GOTO   80
140   STOP
150   PRINT  "Z";
160   GOTO   20
170   PRINT  "D";
180   GOTO   120
```

Jaki będzie efekt wykonania tego programu?

Wykorzystywanie instrukcji warunkowej. Instrukcja warunkowa (IF...THEN) jest najważniejszą instrukcją w sterowaniu wykonywania programu. Jest ona też najczęściej wykorzystywaną, a więc zarówno efektywność jak i elegancja programu są w dużej mierze od niej zależne.

Musimy pamiętać, że na razie dysponujemy instrukcją, za pomocą której w zależności od wartości warunku możliwe jest: ominięcie wykonania sekwencji instrukcji, włączenie do sekwencji jednej instrukcji lub przejście do wykonywania kolejnej iteracji. Właściwości te powinny być już uwzględnione na etapie projektowania algorytmu – przez takie przekształcanie jego struktury, aby możliwe było zastosowanie instrukcji warunkowej. Należy też pamiętać o istnieniu w instrukcji warunkowej instrukcji przejścia GOTO, a ta nie powinna być używana w nadmiarze.

W definiowaniu warunków należy uwzględniać wszystkie możliwe kombinacje – nawet i te z pozoru niemożliwe. Przy zapisie iloczynu lub alternatywy warunków, najpierw powinny być obliczone warunki, które z największym prawdopodobieństwem dla iloczynu dają wartość *prawda*, a dla sumy – wartość *fałsz*.

Instrukcje cyklu wyliczanego (FOR...NEXT) są podstawowe w realizacji warunku strukturalizacji programu. Stąd też staranne dobranie identyfikatora zmiennej sterującej – nie kolidującego z innymi oraz przejrzystej postaci wyrażeń parametrów wartości cyklu – jest bardzo istotne.

Dla jasności w zrozumieniu struktury cyklu nie wolno:

1. Włączać do sekwencji instrukcji w cyklu instrukcji znajdujących się poza zakresem numerów linii instrukcji FOR i NEXT (z wyjątkiem wywołań podprogramów i funkcji definiowanych – ale o tym potem).
2. Wykorzystywać instrukcji zawartych w sekwencji instrukcji w cyklu inaczej jak poprzez instrukcję początku cyklu FOR.

Elegancki zapis tekstu programu. Nadszedł już czas, aby wskazać pewne, umowne zasady zapisu tekstu programu. Stosowanie tych zasad przyniesie korzyść w postaci spełnienia potrzeb estetycznych, a co ważniejsze znacznie ułatwi zrozumienie struktury programu przez siebie (po kilku tygodniach z nim rozłąki) oraz przez innych, którym go udostępnimy. A więc *ad rem:*

1. Oczywiście w tekst programu należy wstawić odpowiednią liczbę instrukcji komentarza.
2. Każda linia powinna być zapisywana z wykorzystaniem odstępów (spacji) między poszczególnymi elementami składni – na pewno przed i po słowie kluczowym. Niektórzy sugerują też otaczanie spacjami każdego operatora – ja pozostawiam to do wyboru programującym.

```
10 FOR i=x+1 TO x*x STEP (x-1)/2

10 FOR  i = x + 1  TO  x * x  STEP  ( x - 1 ) / 2
```

3. Dla zaznaczenia fragmentów struktur programu warto stosować dodatkowe instrukcje komentarza wskazujące początek i koniec logicznie wyznaczanej struktury oraz wcięcia w zapisie instrukcji do niej należącej. Interpretację tej zasady pokażę na kilku przykładach.

```
10 IF  x>0  THEN  40          'Pokazany jest zakres instrukcji warun-
20   y = (x+1) / (x-1)        'kowej - obejmujacy instrukcje pomijane
30   n = n - 1                'jezeli warunek jest spelniony.
40 REM ENDIF
```

```
10 IF  x>0  THEN  50          'Instrukcja warunkowa zawiera dwie alter-
20 REM ELSE                   'natywy - pierwsza oznaczona przez ELSE
30   y=(-x)*LOG(-x)           'jezeli warunek nie jest spelniony,
40   GOTO  70
50 REM THEN                   'oraz druga znajdujaca sie po THEN jest
60   y=x*EXP(x)               'spelniony.
70 REM ENDIF
```

Uwaga: Podany tutaj strukturalny zapis tej instrukcji odbiega od ogólnie przyjętego standardu rozszerzonej funkcjonalnie instrukcji warunkowej, która ma postać

if *warunek*
then
  sekwencja_instrukcji_jeżeli_warunek_spełniony
else
  sekwencja_instrukcji_jeżeli_warunek_niespełniony
end

```
990   GOTO  2000
1000  REM FROM  200,320       'Informacja, z ktorej linii mozе nastapic
1010  '...                    'przejscie do linii numer 1000.
```

```
200 REM WHILE  y<ym                'Instrukcja iteracyjna
210   IF  y>=ym  THEN 250          'z warunkiem poczatkowym
220   LET x=x+n
230   LET y=x*LOG(x) + x*x
240   GOTO  200                    'powtorzenie cyklu
250 REM END WHILE                  'domkniecie instrukcji cyklu
```

```
490 LET s=0
500 REM REPEAT                     'Instrukcja iteracyjna
501                                'z warunkiem koncowym
510   INPUT  x                     '   instrukcje w cyklu wyko-
520   LET  s=s+x                   '   naja sie przynajmniej raz
530   IF  x<>0  THEN  500          'powtorzenie cyklu
540 REM UNTIL  x<>0                'domkniecie instrukcji cyklu
```

Uwaga: Podane tutaj strukturalne zapisy instrukcji iteracyjnych zawierają w komentarzach słowa kluczowe, które są wykorzystywane w standardowych instrukcjach iteracyjnych posiadających postać

```
while warunek do                    'wykonuj instrukcje w cyklu
  instrukcje_w_cyklu                'dopóki warunek jest spełniony
end;
repeat                              'wykonuj instrukcje w cyklu
  instrukcje_w_cyklu                'tak długo, jak długo warunek
until warunek;                      'nie jest spełniony
```

```
680 LET s=0
690 INPUT  n,k
700 FOR i=1 TO n STEP k          'Instrukcja iteracyjna
710   INPUT  x                   'z wyliczaniem.
720   LET  s=s+x*k
730 NEXT i
```

Sytuacje błędne

Undefined line

Niezdefiniowany numer linii występujący w instrukcji warunkowej lub przejścia. Inaczej mówiąc w programie brakuje linii, do której należy przejść w wyniku zmiany sekwencji wykonywania instrukcji programu. Instrukcja, której numer linii jest podawany w instrukcji warunkowej lub przejścia może być też instrukcją komentarza (lub nawet instrukcją danych – jest to naruszenie dobrego stylu programowania). W takim przypadku wykonywana będzie kolejna instrukcja z sekwencji. Przyczyną powstania takiego błędu są najczęściej pomyłki we wprowadzaniu tekstu programu lub też w trakcie jego poprawiania usunęliśmy – może niepotrzebnie? – daną instrukcję. Należy też pamiętać, że często poprawiając jakiś fragment programu, wykorzystywaliśmy go też z innego miejsca programu. Wystąpienie tego błędu powinno mobilizować nas do przejrzenia sterowania całego programu.

NEXT without FOR
FOR without NEXT

Niezgodność w parze instrukcji początku i końca cyklu. Przyczyna tego błędu jest spowodowana brakiem jednej z dwóch instrukcji tworzących cykl, pomyłką w zapisie zmiennej sterującej w jednej z tych instrukcji lub złą organizacją sterowania w programie. W tym trzecim przypadku została naruszona zasada dostępu do instrukcji wewnątrz cyklu wyliczonego jedynie poprzez instrukcję początku cyklu. Inaczej mówiąc wystąpiło bezpośrednie przejście z zewnątrz cyklu do jego wnętrza. Dla uniknięcia tego rodzaju błędów należy bezwzględnie stosować się do zasad podanych w tym rozdziale. Również i tutaj wystąpienie tego błędu powinno mobilizować nas do przejrzenia całej struktury sterowania w programie.

Problemy

6.1. Podaj wartości wyrażeń relacji

```
a)  ABS(x) >= x                    e)  SIN(x) <= 1
b)  LOG(y)=0    ,gdy y<>1          f)  x^2 + y^2 > 1    ,gdy x=1.2
c)  x^2 + 4 <> (x+2)^2             g)  a < 1            ,gdy a=-2
d)  SQR( b*b - 4*a*c ) > 0         h)  x^.5 = SQR(x)
```

6.2. Podaj jakim tekstem należy uzupełnić miejsca oznaczone *a, *b, *c i *d w tekście programu

```
10   INPUT  x,y
20   LET  od=(x-3)^2 + (y-2)^2
30   IF  x*x + y*y < 16  THEN  100
40     IF  od<25  THEN  70
50       PRINT  "Punkt  *a"
60       END
70     PRINT  "Punkt  *b"
80     END
100  IF  od<25  THEN  130
110    PRINT  "Punkt  *c"
120    END
130  PRINT  "Punkt  *d"
140  END
```

6.3. Zapisz w BASICu fragmenty programów opisanych za pomocą sieci działań pokazanych na rys. 6.14. Jeżeli jest to konieczne, dokonaj reorganizacji sieci działań.

6.4. Zapisz w BASICu programy realizujące algorytmy opracowane w postaci sieci działań dla problemów z rozdziału 1. Przeanalizuj wszystkie konieczne reorganizacje struktur sieci działań dla uzyskania lepszej struktury programu.

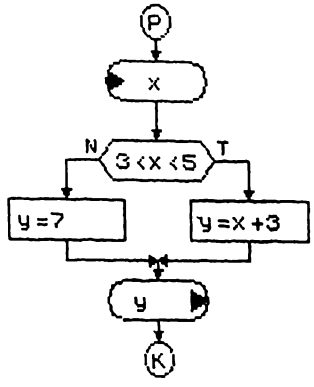
6.5. Podaj wyniki wykonania programów

```
a)  10 LET n=0
    20 REM REPEAT
    30   IF  n<5  THEN  50
    40     PRINT  n;
    50   REM ENDIF
    60   LET n=n+1
    70   IF  n<10  THEN  20
    80 REM UNTIL  n<10
```

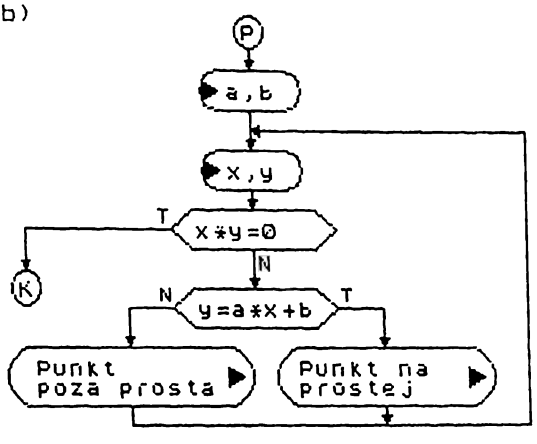
```
b)  10 LET  r=1
    15 LET s=1
    20 REM WHILE  s<=20
    25   IF  s>20  THEN  55
    30   LET  p=s+r
    35   LET  r=s
    40   LET  s=p
    45   PRINT  s;
    50   GOTO  20
    55 REM END WHILE
```

```
c)  10 FOR i=1 TO 10
    20   LET  k=i
    30   REM REPEAT
    40     LET  k=k*2
    50     PRINT  k;
    60     IF  k<i  THEN  30
    70   REM UNTIL  k<i*i
    80 NEXT i
```

a)

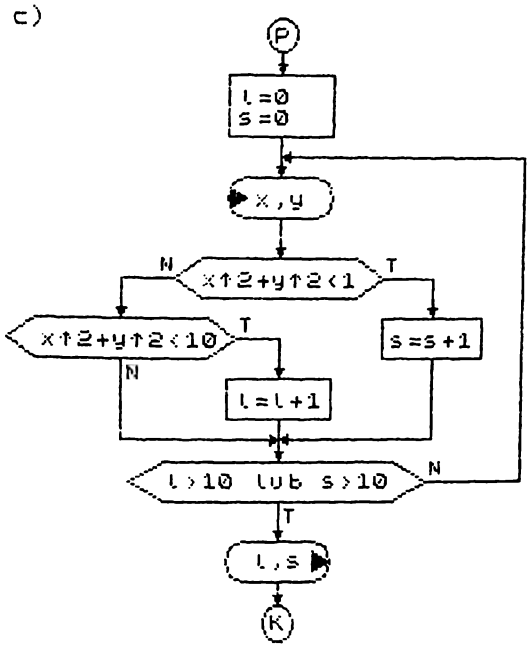


b)

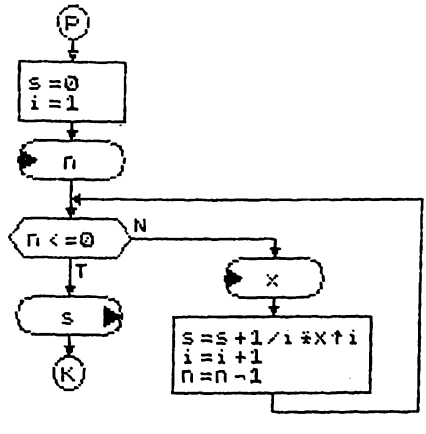


Rys. 6.14 (część 1)

c)



d)



Rys. 6.14 (część 2)

6.6. Napisz programy obliczania sumy wyrazów ciągu:

a) szeregu arytmetycznego złożonego z n liczb naturalnych (n jest rzędu kilku tysięcy) i porównaj czas wykonania tego programu z wykonaniem programu liczącego sumę n liczb ze wzoru Gaussa

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$$

Wskazówka: Dla zmierzenia czasu wykonywania programów użyj zegarka ze stoperem. Drugi program umieść w instrukcji cyklu wielokrotnego dla uzyskania odpowiedniej dokładności miary czasu.

b) szeregu potęgowego

$$\sin x = \sum_{i=0}^{n} (-1)^i \frac{x^{2i+1}}{(2i+1)!}$$

$$e^x = \sum_{i=0}^{n} \frac{x^i}{i!}$$

i porównaj wyniki dla $n = 10, 100, 1000$ z wartościami uzyskanymi za pomocą funkcji SIN(x) i EXP(x).

6.7. Napisz program obliczania współczynników równania liniowego $y = ax + b$ dla dwóch n-elementowych ciągów wartości x_i i y_i korzystając ze wzorów:

suma wartości $\quad sx = \sum_{i=1}^{n} x_i \qquad sy = \sum_{i=1}^{n} y_i$

suma kwadratów $\quad kx = \sum_{i=1}^{n} x_i^2 \qquad ky = \sum_{i=1}^{n} y_i^2$

suma iloczynu $\quad xy = \sum_{i=1}^{n} x_i y_i$

średnia $\quad rx = sx/n \qquad ry = sy/n$

odchylenie standardowe $\quad dx = kx/n - sx^2 \qquad dy = ky/n - sy^2$

Współczynniki równania

$$a = \frac{xy - n * rx * ry}{kx - n * rx}$$

$$b = ry - a * rx$$

współczynnik korelacji

$$r_0 = \frac{xy - rx * ry}{dx^2 - dy^2}$$

6.8. Podaj jakie błędy popełniono w podanych przykładach programów.

a)
```
10 IF  a<b  THEN  10
20   PRINT a
30 REM ENDIF
```

b)
```
10 IF  x  THEN  30
20   PRINT  SGN(x)
30 REM ENDIF
```

c)
```
100 LET x=.5
110 REM REPEAT
120   LET x=x$x
130   IF  x<1  THEN  110
140 REM UNTIL  x<1
```

d)
```
100 LET x=.5
110 REPEAT
120   PRINT  x^2+1
130   LET x=x+.01
140 UNTIL  x<1
```

e)
```
100 FOR i=-5 TO 10 STEP -1
110   PRINT  i
120 NEXT i
130 '...
```

f)
```
100 FOR x=0 TO 100
110   LET  x=x+10
120   PRINT x
130 NEXT
```

g)
```
100 LET z=1
210 READ  y
220 LET z=z + y$y
230 LET z=z/8
240 IF  y<>0  THEN  210
250 PRINT  x
260 IF  x<100  THEN 230
270 PRINT
280 END
300 DATA 16,31,50,60,30,0
```

h)
```
200 FOR i=1 TO 100
210   LET s=0
220   FOR j=1 TO i/2
230     GOTO 260
240   NEXT j
250   PRINT  s
260   LET s=s + j$j
270   GOTO  240
280 NEXT i
```

Let's learn BASIC English

condition warunek
relation relacja
less mniejsze; ~ **than** nie większe
greater większe; ~ **than** nie mniejsze
equal równe; **non** ~ nierówne
true prawda
false fałsz

cycle, loop cykl; pętla
value wartość; **initial** ~ wartość początkowa; **terminal** ~ wartość końcowa; **step** ~ wartość kroku
steering variable, control variable zmienna sterująca
IF...THEN... JEŻELI...WTEDY...
GOTO PRZEJDŹ DO
FOR...TO...STEP... DLA...DO...KROK...
NEXT NASTĘPNE
IF a > b THEN GOTO 50 JEŻELI wartość zmiennej a jest większa od wartości zmiennej b TO PRZEJDŹ DO instrukcji z linii numer 50
FOR i = 1 TO n STEP k DLA wartości od 1 do wartości zmiennej *n* co *k* wartości wykonaj

END of lesson

7. Struktury tablicowe

Dotychczas w programie do przechowywania wartości znamy tylko zmienne proste typu liczbowego lub tekstowego. Każda z nich ma swój unikalny identyfikator tworzący nazwę zmiennej. Jeżeli teraz zadaniem programu jest wykonanie takich samych czynności dla różnych zmiennych, to musimy niestety dla każdej z nich zapisać osobne instrukcje z argumentami ich dotyczącymi. Rozwiązanie takie jest mało efektywne i dość czasochłonne.

W językach wysokiego poziomu, a więc również w języku BASIC, wprowadzono złożoną strukturę danych zwaną tablicą – grupującą pod wspólnym identyfikatorem zbiór wartości tego samego typu.

W rozdziale tym zostaną zaprezentowane zasady deklaracji tablic oraz sposoby korzystania z nich w programowaniu.

7.1. Deklarowanie tablic

Zanim przedstawię formalnie zapis tablicy w programie, wyobraźmy sobie pudełko zawierające przegródki. Każda z nich może zawierać tylko pojedynczą zawartość. Możemy więc mieć pudełka z jednym rzędem przegródek, pudełka z wieloma rzędami oraz wreszcie pudełko z wieloma warstwami, z których każda zawiera wiele rzędów przegródek. Dla wygody, liczba przegródek w każdym rzędzie jest identyczna, a także liczba przegródek w każdej warstwie jest identyczna. Również dla wygody odszukania właściwej przegródki, miejsce jej położenia identyfikujemy odpowiednio:

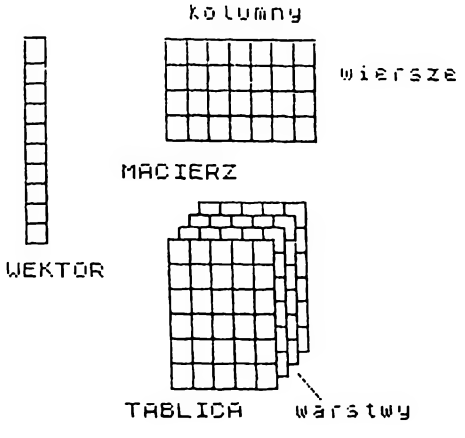
a) numerem w rzędzie,
b) numerem rzędu i numerem w rzędzie,
c) numerem rzędu, numerem w rzędzie i numerem warstwy.

Powróćmy teraz do pojęć z zakresu języka programowania.

Tablicą jest więc struktura danych grupująca zbiór wartości tego samego typu – pudełko, w którym każda z przegródek jest elementem tej tablicy.

Tablicą jednowymiarową zwaną *wektorem* jest pojedyncza *kolumna* elementów – pudełko z jednym rzędem przegródek. Numer elementu w kolumnie nazywamy *indeksem* elementu.

Tablicą dwuwymiarową zwaną *macierzą* jest wiele kolumn złożonych z tej samej liczby elementów – pudełko z kilkoma rzędami przegródek. Wszystkie elementy o tym samym indeksie z każdej kolumny tworzą *wiersz*. Numerem elementu w macierzy jest para indeksów *wiersza* i *kolumny*. Dalej jest już prosto. Możemy mieć *tablicę trójwymiarową* złożoną z wielu kolumn i *warstw*. Numerem elementu w tej tablicy jest trójka indeksów – *wiersz*, *kolumna* i *warstwa*. W niektórych implementacjach języka BASIC możliwe są działania na tablicach cztero- i więcej wymiarowych. Rysunek 7.1. przedstawia graficznie przykładowy wektor, macierz i tablicę trójwymiarową.



Rys. 7.1

Tablice mogą grupować wartości typu liczbowego (tablice typu liczbowego) oraz wartości typu tekstowego (tablice typu tekstowego).

Deklaracja zmiennych prostych jest niejawna. Przy wprowadzaniu pojęcia zmiennej zaznaczyliśmy, że oprócz nazwy, wartości określonego typu, zmienna może mieć również atrybuty.

Atrybutami *zmiennej tablicowej* jest liczba *wymiarów* oraz *rozmiar* każdego z wymiarów określony liczbą elementów. Atrybuty te służą systemowi BASIC do zarezerwowania odpowiedniej liczby słów pamięci na przechowywanie wartości zgrupowanych w tablicę. Deklaracja tablicy powinna więc być jawna, gdyż tylko programista wie, ile jest tych wartości i jak mają być przechowywane.

Deklaracji tablicy dokonuje się za pomocą *instrukcji deklaracji.*

DIM *identyfikator* (*rozmiar_k*)
DIM *identyfikator* (*rozmiar_k*, *rozmiar_w*)
DIM *identyfikator* (*rozmiar_k*, *rozmiar_w*,..., *rozmiar_n*)

Identyfikatorem zmiennej tablicowej (tablicy) jest identyfikator tworzony analogicznie jak dla zmiennych prostych. Nazwą tablicy typu liczbowego jest więc pojedyncza litera, ewentualnie uzupełniona literą lub cyfrą. Nazwą tablicy typu tekstowego jest identyfikator uzupełniony symbolem dolara $. Identyczne identyfikatory dla zmiennych prostych oraz tablic typu liczbowego i tekstowego o różnych wymiarach oznaczają różne obiekty.
W nawiasach jest podawany *rozmiar* danego wymiaru tablicy. Rozmiar jest określony dodatnią wartością wyrażenia całkowitego podającego maksymalną wartość indeksu elementu w danym wymiarze. Minimalna wartość indeksu elementu jest standardowo równa 0. Odpowiednio

rozmiar_k – określa liczbę elementów w kolumnie (wektorze),
rozmiar_w – określa liczbę elementów w wierszu (liczbę kolumn),
rozmiar_n – określa liczbę elementów w *n*-tym wymiarze.

W jednej instrukcji deklaracji można deklarować kilka tablic, oddzielając deklaracje przecinkami.

Wyrażenie całkowite określające rozmiar może być dowolnym wyrażeniem typu liczbowego, o wartości zaokrąglonej (w niektórych implementacjach o części całkowitej) do wartości całkowitej. Najprościej jest podawać rozmiar bezpośrednio w postaci stałej będącej liczbą naturalną.

Instrukcje deklaracji:

```
10 DIM v(9),a$(24),c1(5,3),c9(3,5)
20 DIM x(10,10),w$(5,0),z(3,4,5)
```

spowodują zadeklarowanie tablic:

v(9) – wektor typu liczbowego, 10-elementowy;
a$(24) – wektor typu tekstowego 25-elementowy;
c1(5,3) – macierz typu liczbowego z 6 wierszami i 4 kolumnami;
c9(3,5) – macierz typu liczbowego z 4 wierszami i 6 kolumnami (porównaj z *c1*(5,3));
x(10,10) – macierz typu liczbowego z 11 wierszami i 11 kolumnami – zwana *macierzą kwadratową*, gdyż liczba wierszy jest równa liczbie kolumn;

w$(5,0) – macierz typu tekstowego z 6 wierszami i 1 kolumną – pomimo istnienia jednej kolumny nie jest to wektor;
z(3,4,5) – tablica trójwymiarowa z 4 wierszami, 5 kolumnami i 6 warstwami.

Liczba elementów tablicy jest równa:

(*rozmiar_k* + 1)*...*(*rozmiar_n* + 1)

Deklaracja tablicy musi wystąpić przed pierwszym odwołaniem się w programie do jej elementów. W przeciwnym razie zostanie dokonana *deklaracja standardowa* tablicy według następujących zasad:

1. Wystąpienie w odwołaniu jednego indeksu jest równoważne zadeklarowaniu wektora o 11 elementach.
2. Wystąpienie w odwołaniu dwóch indeksów jest równoważne zadeklarowaniu macierzy o 11 wierszach i 11 kolumnach.
3. Wystąpienie trzech i więcej indeksów jest równoważne zadeklarowaniu tablicy o tej samej liczbie wymiarów po 11 elementów w każdym wymiarze.

7.2. Zmienne indeksowane

Każdy element tablicy jest dostępny do odczytu lub zapisu wartości jako *zmienna indeksowana* typu odpowiadającego typowi tablicy.

Nazwa zmiennej indeksowanej jest złożona z identyfikatora tablicy i symbolu dolara $ dla typu tekstowego oraz z tylu indeksów w nawiasach, z ilu wymiarów składa się tablica. Poszczególne indeksy są rozdzielone przecinkiem. Indeks jest wyrażeniem całkowitym. Wartość tego wyrażenia musi być liczbą nieujemną i nie może przekraczać wartości rozmiaru tablicy w danym wymiarze.

Po uruchomieniu programu, po deklaracji tablicy wartości początkowe wszystkich zmiennych są równe zeru.

Dla wektora z rys. 7.2. i deklaracji

10 DIM *v*(8)

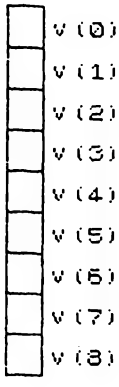
mamy do dyspozycji 9 zmiennych indeksowanych typu liczbowego o nazwach:

v(0), *v*(1), *v*(2), *v*(3), *v*(4), *v*(5), *v*(6), *v*(7), *v*(8)

Podobnie dla macierzy z rys. 7.3 i deklaracji

10 DIM *c$*(2,4)

mamy do dyspozycji (2 + 1)*(4 + 1) = 15 zmiennych indeksowanych typu tekstowego o nazwach:



DIM v(8)

Rys. 7.2

c$

0,0	0,1	0,2	0,3	0,4
1,0	1,1	1,2	1,3	1,4
2,0	2,1	2,2	2,3	2,4

DIM c$(2,4)

Rys. 7.3

c$*(0,0), *c$(0,1), *c$*(0,2), *c$*(0,3), *c$*(0,4)
c$*(1,0), *c$(1,1), *c$*(1,2), *c$*(1,3), *c$*(1,4)
c$*(2,0), *c$(2,1), *c$*(2,2), *c$*(2,3), *c$*(2,4)

Zmienne indeksowane mogą być wykorzystywane w wyrażeniach zamiast zmiennych prostych. Mogą też być zmiennymi przypisywanymi w instrukcji przypisania oraz mogą występować na listach zmiennych. Zmienna indeksowana typu liczbowego (a tym bardziej tekstowego) nie może być zmienną sterującą instrukcji cyklu wyliczanego.

Rozważmy jeszcze deklarację

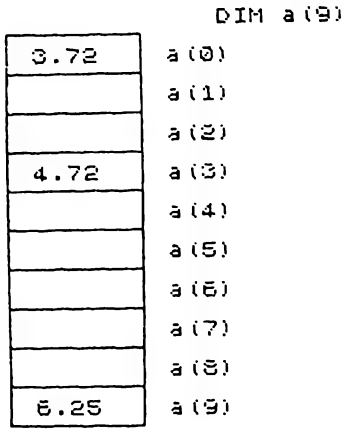
10 DIM *p*(3,7)

oraz dla i = 4 przykładowe zmienne indeksowane:

p(i−1, i*2−1) – dotyczy elementu *p*(3,7)
p(i−3, SQR(i)) – dotyczy elementu *p*(1,2)
p(i) – dotyczy elementu *p*(5), ale z wektora zadeklarowanego standardowo jako DIM *p*(10) (11 elementów)

7.3. Działania na wektorach

W programowaniu najczęściej korzystamy z tablic jednowymiarowych – wektorów. Tutaj przedstawię podstawowe, prawie już standardowe działanie na wektorach. Rozważmy instrukcje działające na wektorze (rys. 7.4),



Rys. 7.4

```
10 DIM a(9)
20 LET a(0)=3.72
30 LET a(3)=a(0)+1
40 LET a(9)=6.25
```

których wykonanie spowoduje kolejno:

10 – zadeklarowanie wektora 10-elementowego;

20 – przypisanie zmiennej indeksowanej o indeksie równym 0 wartości 3.72;

30 – przypisanie zmiennej indeksowanej $a(3)$ wartości $a(0)$ zwiększonej o wartość 1, a więc wartości 4.72;

40 – przypisanie zmiennej indeksowanej $a(9)$ wartości 6.25; zapis $a(9)$ tylko w instrukcji deklaracji dotyczy liczby elementów, w pozostałych przypadkach jest to dziesiąty (licząc od zera) element wektora.

Dostęp do kolejnych elementów wektora uzyskuje się korzystając z instrukcji cyklu wyliczanego w postaci

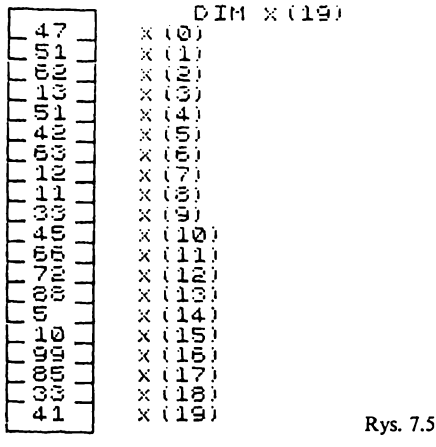
```
10 DIM  w(n)
...
100 FOR i=0 TO n
...REM  dostep do w(i)
200 NEXT i
```

W instrukcji kolejne wartości zmiennej sterującej są indeksami kolejnych elementów wektora, czyli w rezultacie wszystkich jego zmiennych indeksowanych.

Wykorzystując podany schemat wczytamy wartości wektora z zestawu danych.

```
190  DIM x(19)
200  FOR i=0 TO 19
210    READ x(i)
220  NEXT i
599  REM  Zestaw danych
600  DATA 47,51,62,13,51,42,63
610  DATA 12,11,33,45,66,72,88
620  DATA  5,10,99,85,33,41
```

Wypełniony wektor jest pokazany na rys. 7.5.



Rys. 7.5

Wymiana instrukcji w linii 210 na instrukcję

210 INPUT *x*(i)

umożliwia nadawanie z klawiatury wartości elementom wektora, przy czym tutaj znak zapytania pojawi się 20 razy.

Z kolei instrukcje

```
500  FOR i=0 TO 19
510    PRINT x(i)
520  NEXT i
```

spowodują wyświetlenie kolumny wartości z kolejnych elementów wektora.

Podobnie instrukcje

```
400  LET s=0
410  FOR i=0 TO 19
420    LET s=s+x(i)
430  NEXT i
```

pozwalają uzyskać sumę wartości zapisanych w wektorze.

Z kolei instrukcje

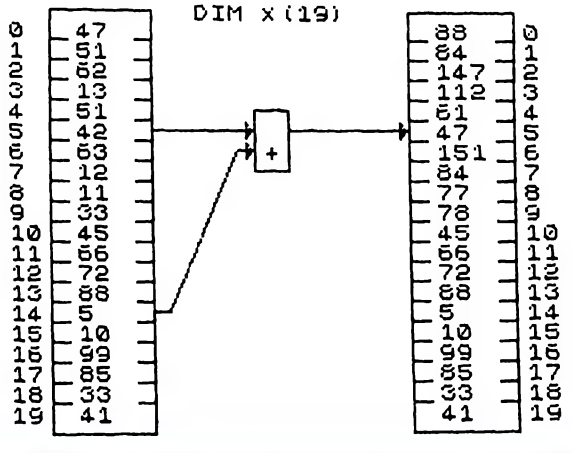
```
450  LET  m=s(0)                    'wartosc maksymalna
455  LET  i=0                       'numer elementu
460  FOR  k=1  TO 19
465    IF m<=s(k) THEN  480         'sprawdzenie z kolejna
466                                 ' wartoscia,
470      LET m=s(k)                 'zmiana aktualnej
475      LET i=k                    ' wartosci maksymalnej,
480    REM ENDIF
485  NEXT k
490  PRINT "Wartosc maksymalna:"; m; "w elemencie nr:"; i
```

prezentują się same.

Przeanalizujmy bliżej funkcjonowanie instrukcji

```
50 FOR i=0 TO 19
55   LET x(i)=x(i)+x(19-i)
60 NEXT i
```

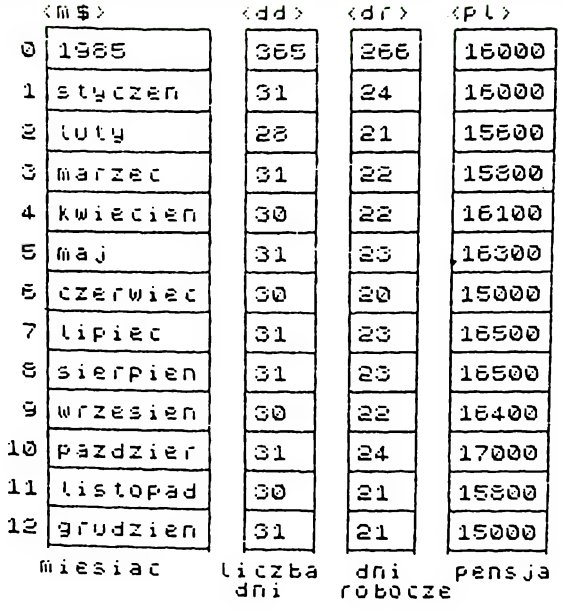
Zbadajmy wykonanie instrukcji z linii 55 dla $i = 5$, co ilustruje rys. 7.6. Widać z niego, że zmiennej o indeksie 5 przypisujemy sumę dotychczasowej wartości i wartości ze zmiennej o indeksie $19-5 = 14$. Dokładniejsza analiza całego fragmentu programu pokazuje, że do wartości kolejnych dziesięciu elementów wektora zostaną dodane wartości pobierane w kolejności odwrotnej z tegoż wektora. Na rysunku 7.6. z lewej strony pokazana jest zawartość pierwotna wektora, a z prawej zawartość po wykonaniu tego fragmentu programu.



Rys. 7.6

Indeks porządkuje kolejność elementów wektora, czyli kolejność zmiennych indeksowanych. Na tej podstawie dla zmiennej indeksowanej można wskazać zmienną ją poprzedzającą i zmienną po niej następującą:

$w(i-1)$ – zmienna poprzedzająca
$w(i)$ – zmienna aktualna
$w(i+1)$ – zmienna następująca
Należy pamiętać o elementach brzegowych nie mających poprzednika lub następnika.



Rys. 7.7

Na rysunku 7.7 pokazano wektory zawierające odpowiednio:
m$ (12) – wykaz miesięcy
da (12) – liczbę dni w każdym miesiącu
dr (12) – liczbę dni roboczych w każdym miesiącu
pl (12) – wartości wypłat pensji pracownika w każdym miesiącu
W tym przypadku indeks elementu pełni rolę wiążącą informacje zapisane w różnych wektorach. Dla uproszczenia elementy o indeksie równym zeru przeznaczono na informacje zbiorcze.

Przykład 7.1. „Przeszukiwanie liniowe"

Opis problemu

W wektorze 1000-elementowym może być przechowywanych do 1000 wartości liczbowych. Wartości te są wprowadzone z klawiatury jako ciągi liczb zakończone wartością równą zeru.

Na przemian z wprowadzaniem użytkownik wskazuje z klawiatury te wartości, które mają być usunięte (użytkownik może żądać też usunięcia wartości nieistniejącej) oraz może zażyczyć sobie wyprowadzenia aktualnej zawartości wektora w kolejności wzrastających wartości.

Analiza problemu

Program, jak łatwo można to zauważyć, powinien składać się z pięciu różnych części:

A. Wybór rodzaju czynności do wykonania.
B. Wstawianie nowej wartości do wektora.
C. Usuwanie wskazanej wartości z wektora.
D. Wyprowadzenie aktualnej zawartości wektora (posortowanej wzrastająco).
E. Stop programu.

Zatrzymajmy się tutaj przy punktach B, C i D. Rozważmy następujące rozwiązania:

*W*1. Wartości wstawiamy w pierwsze wolne miejsce w wektorze (oznaczone wartością równą 0). Przy usuwaniu wartości przeglądamy kolejno elementy wektora i po znalezieniu danej wartości pozostawiamy w danym elemencie wartość równą zeru. Przy wyprowadzaniu wyszukujemy w wektorze kolejno elementy o wartościach wzrastających.

*W*2. Wartości wstawiamy zawsze na końcu aktualnie zapełnionej części wektora. Po znalezieniu wartości do usunięcia, pozostałą część wartości istotnych przesuwamy, odzyskując na końcu części zapełnionej wolne miejsce. Przed wprowadzeniem sortujemy (wzrastająco) zapełnioną część wektora, a następnie wyprowadzamy kolejne wartości.

*W*3. Przy ustawianiu wartości wyszukujemy w zapełnionej części wektora takie miejsce na nową wartość, aby po rozsunięciu i wstawieniu wartości były one zawsze uporządkowane wzrastająco. Przy usuwaniu wartości pozostałą zapełnioną część wektora dosuwamy. Wartości wyprowadzamy kolejno z zapełnionej części wektora.

Dokonajmy teraz bardzo przybliżonego oszacowania złożoności każdego z rozwiązań. Przyjmiemy, że analizujemy stan po dłuższym okresie pełnego funkcjonowania każdego z algorytmów oraz że był już przypadek całkowitego wypełnienia wektora. W wektorze znajduje się n ($n < 1000$) wartości.

	$W1$	$W2$	$W3$
średnia liczba dostępów do wektora dla znalezienia wolnego miejsca na nową wartość	$\frac{1000}{1000-n}$ [a]	1 [b]	$\left(\frac{1}{2}n+1\right)+\left(\frac{1}{2}n\right)*2$ [c]
średnia liczba dostępów do wektora dla znalezienia elementu istniejącego	500 [d]	$\left(\frac{1}{2}n\right)+\left(\frac{1}{2}n\right)*2$ [e]	$\left(\frac{1}{2}n\right)+\left(\frac{1}{2}n\right)*2$ [f]
średnia liczba dostępów do wektora dla stwierdzenia braku elementu	1000 [g]	n [h]	$\frac{1}{2}n+1$ [i]
średnia liczba dostępów do wektora dla wyprowadzenia wartości w kolejności rosnącej	n*1000 [j]	? [k]	n [l]

Objaśnienia:
a) miejsca wolne (jest ich 1000-n) są rozłożone równomiernie, b) pierwsze wolne miejsce jest znane, c) należy przejrzeć co najmniej połowę plus jeden zapełnionej części wektora, a potem jeszcze przesunąć pozostałą część wektora (co wymaga dwóch dostępów do wektora na każde przesunięcie pojedynczego elementu), d) połowa całego wektora, e) wystarczy przejrzeć połowę, a potem resztę przesunąć, f) analogicznie jak e), przy zastosowaniu algorytmu przeszukiwania liniowego (istnieją inne algorytmy wyszukujące przy $\log n$ dostępach – przeszukiwanie binarne), g) niestety cały wektor, h) tylko część zapełnioną, i) tylko połowę części zapełnionej, jeżeli bowiem kolejna wartość jest większa od poszukiwanej to znaczy, że tej poszukiwanej nie ma, j) tutaj można uprościć algorytm, ale i tak będzie on dość czasochłonny, k) zależnie od wybranej metody sortowania (np. $n*\log n+1$), l) wyprowadzamy od razu kolejno n wartości.

Wersję $W1$ odrzucamy. Pozostaje nam wybór między rozwiązaniem $W2$ a $W3$. Należy zauważyć, że po wykonaniu sortowania w wersji $W2$, mamy sytuację równoważną sytuacji między operacjami w wersji $W3$. Wydaje się, że teraz ostateczny wybór zależy od tego jak często mamy następujące sytuacje:

a) wielokrotne wstawianie, bardzo rzadkie usuwanie i wyprowadzanie oraz żądania usunięcia elementów nieistniejących;
b) częste wstawianie wymiennie z usuwaniem, a rzadkie żądania wyprowadzenia czy usunięcia elementów nieistniejących;
c) bardzo częste wstawianie i żądanie wyprowadzenia zawartości elementów, a rzadkie usuwanie;
d) częste wstawianie elementów oraz częste żądanie usunięcia elementów, w tym szczególnie dużo żądań usunięcia elementów nieistniejących.

Dwie pierwsze sytuacje skłaniają nas do wyboru rozwiązania w wersji $W2$, a dla pozostałych rozwiązanie w wersji $W3$. Wybieramy do rozwiązania algorytm w wersji $W3$.

Metoda rozwiązania

Część A programu zrealizujemy standardowo, prezentując menu programu, na podstawie którego użytkownik będzie wskazywał którą z czynności ma wykonywać program.

Część B programu dokonuje wstawienia nowej wartości. Jeżeli nie byłoby warunku uporządkowania (jak w wersji *W2*), to podstawowy zapis tego fragmentu może być następujący:

```
200  INPUT  v
210  LET    m = m+1
220  LET    w(m)=v
```

Tutaj jednak musimy przeglądnąć wektor wartości szukając takiego pierwszego elementu, którego wartość jest większa od wprowadzonej nowej wartości. Element ten będzie bowiem miejscem dla nowej wartości. Zanim więc wartość tę wstawimy, musimy kolejno resztę wartości przesunąć o jedną pozycję. Jest to algorytm przeszukiwania liniowego (dla wygody element w (0) przeznaczymy na aktualną liczbę zajętych elementów wektora). Podobnie realizujemy część C. Wyprowadzanie wartości w części D jest bardzo proste.

Dane wejściowe

Wybór numeru czynności programu według menu — liczba naturalna z przedziału 1...4.
Ciąg wartości wprowadzonych lub usuwanych jest zakończony wartością równą zeru.

Tekst programu

```
10  REM  Program modyfikacji zawartosci
20  REM    wektora z uporzadkowaniem
30  '    ------------------------------
40 DIM  w(100)
50 LET  m=0
100 PRINT "MENU PROGRAMU"
105 PRINT
110 PRINT "1. Wstawienie nowych wartosci"
120 PRINT "2. Usuwanie wartosci"
130 PRINT "3. Wyprowadzanie zawartosci wektora"
140 PRINT "4. Stop programu"
145 PRINT
150 PRINT "Wybierz czynnosc programu"
155 PRINT "podajac jej numer"
160 INPUT c
165 IF  c=1  THEN  200
170 IF  c=2  THEN  400
175 IF  c=3  THEN  600
180 IF  c=4  THEN  700
190 PRINT "Czynnosci sa ponumerowane od 1 do 4"
195 GOTO 100
199 '    -------------------------
200 'REM  Wstawienie nowej wartosci
205 '
210 '    m - liczba elementow tablicy
215 '
220 PRINT "Wprowadzaj kolejno wartosci"
225 PRINT "(wartosc =0 konczy wprowadzanie)
230 LET m=w(0)
235 IF  m=100  THEN  305
240 INPUT v                    'wczytaniewartosci
245 IF  v=0  THEN  150
250 FOR i=1 TO m
```

```
255   IF  w(i)>v  THEN  270   'poszukiwanie miejsca
260 NEXT i
265 LET i=m+1
270 FOR j=m TO i STEP -1       'przesuniecie czesci
275   LET w(j+1)=w(j)          ' wektora
280 NEXT j
285 LET w(i)=v                 'wstawienie
295 LET w(0)=w(0)+1
300 GOTO  230
305 PRINT "Wektor wypelniony w calosci !!"
310 GOTO 150
399 '    ---------------------------
400 REM Usuwanie wartosci
405 '
410 PRINT "Podawaj kolejno wartosci"
415 PRINT " do usuniecia"
420 PRINT "(wartosc =0 konczy usuwanie)
425 LET m=w(0)
430 INPUT v
435 IF  v=0  THEN  150
440 FOR i=1 TO m               'wyszukanie wartosci
445   IF   w(i)=v  THEN  465
450 NEXT i
455 PRINT "Brak takiej wartosci"
460 GOTO  430
465 FOR j=i+1 TO m
470   LET  w(j-1)=w(j)
475 NEXT j
480 LET  w(0)=w(0)-1
485 GOTO  425
599 '    ---------------------------
600 REM  Wyprowadzenie zawartosci wektora
605 '
610 PRINT "Zawartosc wektora"
615 FOR i=1 TO w(0)
620   PRINT w(i),
625   IF  INT(i/5)*5=i  THEN PRINT
630 NEXT i
635 PRINT
640 GOTO 150
699 '    ---------------------------
700 REM  Stop programu
705 '
710 PRINT "Koniec programu"
715 STOP
```

7.4. Działania na macierzach

Macierze są tablicami dwuwymiarowymi. Położenie każdego elementu jest określone dwiema współrzędnymi podającymi numer wiersza i kolumny. W pamięci operacyjnej macierz jest zapisywana kolejno kolumnami.

Dostęp do kolejnych elementów macierzy uzyskuje się korzystając z podwójnej, zagnieżdżonej w sobie instrukcji cyklu wyliczonego o postaci

```
10 DIM  t(m,n)

100 FOR  i=0  TO  m
105   FOR  j=0  TO  n

...REM dostep do t(i,j)

195   NEXT j
200 NEXT i
```

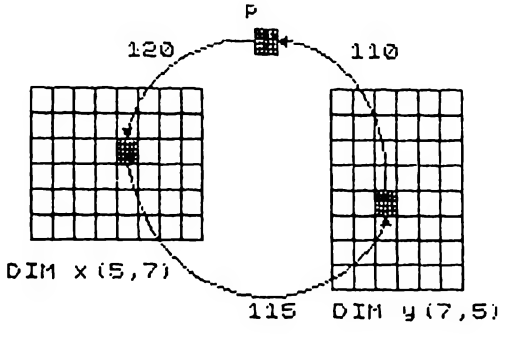
Dostęp do poszczególnych elementów macierzy odbywa się kolejno w wierszach, począwszy od wiersza zerowego.

W tablicy kwadratowej, o takiej samej liczbie wierszy i kolumn, jest pożyteczny dostęp do elementów leżących na przekątnych

```
10  DIM  k(n,n)
...
100  FOR  i=0  TO  n
110    LET  k(i,i)=5
120    LET  k(i,n-i)=7
130  NEXT i
```

Pytanie dla Czytelnika: Jaka będzie zawartość zmiennej indeksowanej $k(3,3)$ dla $n = 6$ po wykonaniu tych instrukcji?

Wykorzystując podany schemat, dokonamy wymiany zawartości między dwoma macierzami $x(5, 7)$ i $y(7,5)$. Podkreślam – macierze te mają jednakową liczbę elementów (rys. 7.8).



Rys. 7.8

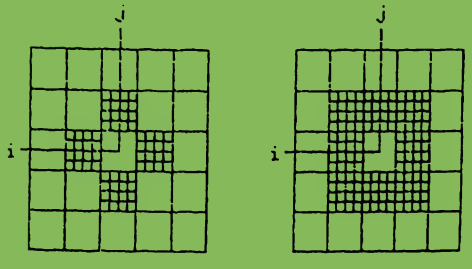
```
10  DIM  x(5,7),y(7,5)
20  '
30  '
100 FOR  i=0  TO  5
105   FOR  j=0  TO  7
110     LET  p=y(j,i)
115     LET  y(j,i)=x(i,j)
120     LET  x(i,j)=p
125   NEXT j
130 NEXT i
```

I jeszcze jedną z tablic wyprowadzimy kolejno wierszami, a drugą – kolejno kolumnami.

```
10  DIM  x(5,7),y(7,5)
20  '
30  '
200 FOR  i=0  TO  5
205   FOR  j=0  TO  7
210     PRINT  x(i,j);
215   NEXT j
220   PRINT
225 NEXT i
```

```
10  DIM  x(5,7),y(7,5)
20  '
30  '
300 FOR  j=0  TO  5
305   FOR  i=0  TO  7
310     PRINT  y(i,j);
315   NEXT i
320   PRINT
325 NEXT j
```

Położenie zmiennej indeksowanej w macierzy jest lokalizowane parą indeksów. Elementami *sąsiednimi* zmiennej indeksowanej o parze indeksów (i, j), są zmienne indeksowane o parach indeksów (rys. 7.9).

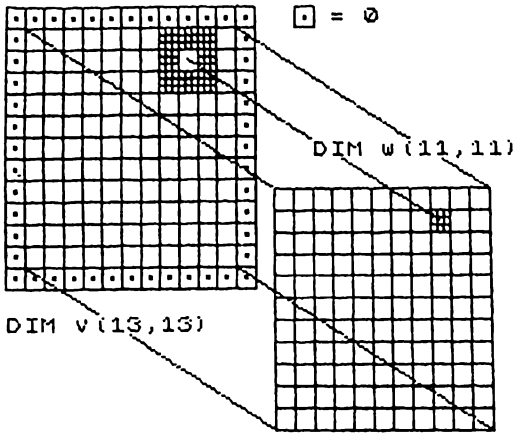


sąsiedztwo otoczenie Rys. 7.9

a) $i-1, j$
b) $i, j+1$
c) $i+1, j$
d) $i, j-1$

Wyróżnia się jeszcze pojęcie *otoczenia* zmiennej indeksowanej tworzone przez 8 zmiennych przyległych do danej. W obu przypadkach należy zwrócić uwagę na ograniczoność sąsiedztwa i otoczenia dla zmiennych indeksowanych będących elementami brzegowymi.

Dla uniknięcia kłopotliwego badania położenia zmiennych indeksowanych przy brzegach tablicy, w przypadku wykorzystywania elementów sąsiednich, stosuje się w programowaniu pewien trik. Rozpatrywaną macierz umieszcza się w tablicy o zwiększonej o dwa liczbie wierszy i kolumn. Elementy brzegowe tej tablicy wypełnia się wartością obojętną dla obliczeń. Rozważmy więc poszukiwanie wartości średniej z wartości elementów otoczenia dla każdego elementu macierzy v z 11 wierszami i 11 kolumnami. Zgodnie z opisem zadeklarujemy dwie tablice – źródłową i z wynikami (rys. 7.10).



Rys. 7.10

```
10  REM  -------DEKLARACJA TABLIC-------------
15  '      v(12,12)  - tablica zrodlowa z danymi
20  '      w(10,10)  - tablica wynikowa
25  '
30  DIM  v(12,12), w(10,10)
35  '
100 REM  -------WCZYTYWANIE TABLICY------------
105 '     zakladamy ,ze po uruchomieniu zawartosc
110 '     tablicy  v  jest poczatkowo rowna  0
115 '
120 FOR i=1 TO 11
125   FOR j=1 TO 11
130      v(i,j)=1
135   NEXT j
140 NEXT i
150 REM ----POSZUKIWANIE SREDNIEJ Z OTOCZENIA---
155 '    opis w tekscie
160 '
165 FOR i=1 TO 11
170   FOR j=1 TO 11
175     LET  s=-v(i,j)
180     FOR k=-1 TO 1
185       s = s + v(i+k,j-1)+v(i+k,j)+v(i+k,j+1)
190     NEXT k
195    LET  w(i-1,j-1)=s/8
200   NEXT j
205 NEXT i
210 REM ------WYPROWADZANIE WYNIKU--------------
215 '
220  FOR i=0 TO 10
225    FOR j=0 TO 10
230      PRINT w(i,j);
235    NEXT j
240    PRINT
245 NEXT i
300 REM  ------------KONIEC---------------------
305 '
310 END
```

W liniach od 175 do 195 zastosowano równocześnie dwie metody obliczenia średniej z sumy wartości elementów otoczenia. W wyrażeniu zapisano sumę wartości trzech elementów z kolumny poprzedzającej, z danej i następującej w stosunku do kolumny o indeksie *j*. Instrukcja cyklu wyliczanego (linie 180–190) powtarza obliczenie dla trzech kolejnych wierszy. Wartość danego elementu o współrzędnych *i*, *j* nie jest brana pod uwagę, stąd wartość początkowa zmiennej s jest równa ujemnej wartości tego elementu. (W programie tym nie uwzględnia się faktu, że elementy brzegowe mają mniej niż 8 sąsiadów w otoczeniu – patrz linia 195).

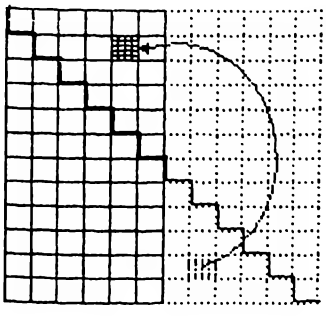
7.5. Tablice trójkątne

Nie zawsze wartości wszystkich elementów tablic (macierzy) są istotne dla obliczeń. Dla przykładu, zapisując w tablicy odległości drogowe między miastami (tablica taka znajduje się w każdym atlasie samochodowym) wystarczy tylko zająć elementy poniżej głównej przekątnej. Z Warszawy do Krakowa jest przecież tyle samo kilometrów, co z Krakowa do Warszawy (przynajmniej tak powinno być).

Tablicą trójkątną jest kwadratowa tablica dwuwymiarowa, w której tylko część poniżej głównej przekątnej zawiera wartości istotne dla obliczeń.

W programowaniu deklaruje się całą tablicę, wykorzystując następnie jej część. Jednak często musimy poszukiwać sposobów zmniejszenia zajętości pamięci operacyjnej. Stąd też zamiast *n*-elementowej tablicy kwadratowej (*n* – powinno być parzyste) deklarujemy tablicę prostokątną o *n*/2 liczbie kolumn.

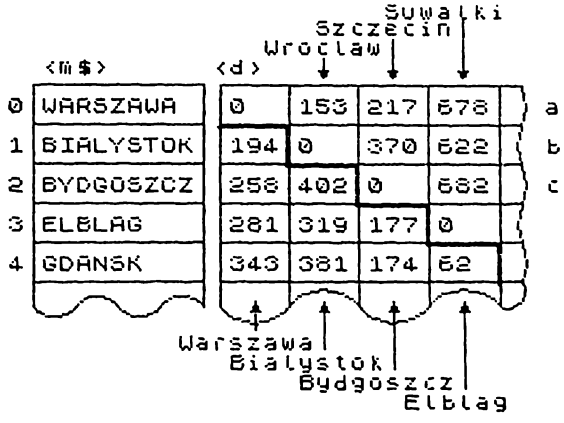
Elementy znajdujące się dotychczas w kolumnach o numerach większych od *n*/2 są umieszczane w prawej górnej części tablicy prostokątnej. Dla $n = 12$ mamy więc sytuację jak na rys. 7.11.



Rys. 7.11

Przykład 7.2. „Odległości drogowe"

Wykorzystanie tablicy trójkątnej pokażę na przykładzie tablicy reprezentującej odległości drogowe między miastami (rys. 7.12)



a - Zielona Gora
b - Wroclaw
c - Szczecin

Rys. 7.12

```
10  REM -----ANALIZA ODLEGLOSCI DROGOWYCH--------
20  '
30  '   Opis zmiennych podstawowych
35  '    m$(19)    - wykaz nazw 20 miast.
40  '    d(19,9)   - tablica prostokatna odwzorowu-
50  '                jaca tablice trojkatna.
55  '
60  '
70  DIM  m$(19) , d(19,9)
75  '
80  REM  -------  --Wczytywanie danych-----------
90  FOR i=0 TO 19
100   READ m$(i)          'wczytanie nazwy miasta
110   LET  k=i
120   FOR j=0 TO i-1
130     LET l=j
140     IF  j<=9  THEN  170
150       LET k = 19-i
160       LET l = 19-j
170     REM ENDIF
180     READ  d(k,l)      'wczytanie odleglosci
190   NEXT j
200 NEXT i
210 REM  ------Wyszukanie par miast lezacych------
220 '          najblizej i najdalej od siebie
230 '
240 ' Nalezy znalezc wartosc najmniejsza i najwieksza
250 ' oraz zapamietac ich wspolrzedne.
260 '
```

```
270 LET d0=1000      'zakladamy ze wszystkie odleglosci
275                  'sa mniejse od  1000 km.
280 LET d9=0         'oczywiscie sa tez wieksze od 0 .
290 FOR i=1 TO 19
300   FOR j=0 TO i-1
310                  'wybranie kolejnej wartosci z tablicy.
320     IF  j>9  THEN  350
330       LET d5=d(i,j)
340       GOTO  370
350     REM ELSE
360       LET d5=d(19-i,19-j)
370     REM ENDIF
380                  'poszukiwanie maksimun.
390     IF  d5<=d9  THEN  440
400       LET d9=d5          'znaleziono wartosc wieksza
410       LET k9=i           'od poprzedniej i jej
420       LET 19=j           'wspolrzedne.
430       GOTO  510
440     REM ELSE
450                  'poszukiwanie minimum.
460       IF  d5>=d0  THEN  500
470         LET d0=d5        'znaleiono wartosc mniejsza
480         LET k0=i         'od poprzedniej i jej
490         LET 10=j         'wspolrzedne.
500       REM ENDIF
510     REM ENDIF
520   NEXT j
530 NEXT i
600 REM  ------Wyprowadzanie wynikow----------
610 PRINT "Miasta polozone najblizej siebie"
620 PRINT "      ";m$(k0);" - ";m$(10);" : " ;d0;"km"
630 PRINT
640 PRINT "Miasta polozone najdalej od siebie"
650 PRINT "      ";m$(k9);" - ";m$(19);" : ";d9;"km"
660 PRINT
670 END

1000 REM  ------Zestaw danych  (fragment)------
1010 DATA "WARSZAWA"
1020 DATA "BIALYSTOK", 194
1030 DATA "BYDOSZCZ" , 258, 402
1040 DATA "ELBLAG"   , 281, 319, 177
1050 DATA "GDANSK"   , 343, 381, 174,  62
1060 DATA "KATOWICE" , 285, 479, 376, 511, 518
1070  ' itd.
```

7.6. Modyfikacje deklaracji tablic

W deklaracji tablicy, zakładaliśmy dotychczas, że standardowo najniższą wartością indeksu elementu w każdym wymiarze jest wartość równa zeru. W wielu jednak zastosowaniach istnienie takiego elementu jest dla nas mylące. Na przykład żądanie podania nazwy czwartego miesiąca spowoduje wyprowadzenie napisu (no właśnie jakiego?) – „Maj" czy „Kwiecień"? Oczywiście możemy pomijać te elementy w tablicy, tylko wtedy po co zajmować miejsce w pamięci operacyjnej. To dość przydługie wprowadzenie było konieczne, aby kolejną instrukcję można było potraktować poważnie. Dla bardziej zaawansowanych wspomnę jeszcze, że na temat minimalnej wartości indeksu tablic odbyło się już kilka poważnych dyskusji naukowych.

OPTION BASE 0
OPTION BASE 1

Wykonanie instrukcji zmienia na wartość 0 lub 1 ustaloną minimalną wartość indeksu dla każdej następnej deklarowanej tablicy. Ustalenie to jest też uwzględnione w deklaracji standardowej tablic. Po uruchomieniu programu zawsze standardowo wykonywane jest OPTION BASE 0.

Dla ilustracji rozważmy program, w którym kolejno występują deklaracje:

```
10  DIM  p(15)          'wektora 16 elementowego
20  DIM  w$(10,1)       'macierzy  11x2=22 elementowej
30  LET  x(5)=10        'standardowo wektora 11 elementowego
40  OPTION BASE 1
50  DIM  r(15)          'wektora 15 elementowego
60  DIM  r$(10,1)       'macierzy  10x1=10 elementowej
70  LET  y(5)=10        'standardowo wektora 10 elementowego
80  OPTION BASE 0
90  DIM  s(15)          'wektora 16 elementowego
100 END
```

W systemie BASIC z interpretatorem instrukcja deklaracji tablicy jest wykonywana zgodnie z kolejnością wykonywanych instrukcji. W tym też momencie następuje dopiero zarezerwowanie odpowiedniego co do wielkości fragmentu pamięci operacyjnej na wartości elementów tablicy. Rozmiary tablicy są zadawane wyrażeniami całkowitymi. Jest więc możliwe na przykład wyznaczenie aktualnego rozmiaru tablicy na podstawie wartości wprowadzonej z zewnątrz. W ten sposób dla każdego przebiegu programu może być wykorzystywana tablica o innych rozmiarach, ale zawsze o tej samej liczbie wymiarów.

Deklaracją dynamiczną struktury danych jest powoływanie struktury w trakcie działania programu i na pewien okres.

Gospodarowanie obszarem pamięci operacyjnej wymaga często usunięcia z niej zbędnych już elementów. Najefektywniejsze jest usuwanie dużych fragmentów zbiorów wartości – jakimi są na przykład tablice. Celowi temu służy *instrukcja wymazywania tablic.*

ERASE *lista_nazw_tablic*

Na *liście_nazw_tablic* mogą być umieszczone, oddzielone od siebie przecinkami, nazwy tablic dotychczas zadeklarowanych jawnie lub standardowo w programie. Wykonanie instrukcji powoduje odzyskanie fragmentu pamięci zajętego dotychczas przez wartości elementów tych tablic.

Instrukcja wymazywania tablic pozwala w jednym programie wykorzystać powtórnie ten sam identyfikator w nowej deklaracji.

Prawidłowy będzie więc zapis programu

```
90   OPTION BASE 1
100  INPUT  "Podaj rozmiar wektora"; n
110  DIM    w(n)
115  '
500      REM  Wykorzystanie wektora n elementowego
995  '
1000 ERASE  w
1010 GOTO   100
```

Przykład 7.3. „Sortowanie bąbelkowe"

Opis problemu

Problem sortowania wartości jest dość częstym elementem w programowaniu (patrz przykład 7.1). Sortowanie polega na uporządkowaniu określonego ciągu elementów według zadanej relacji – najczęściej relacji większości. W najprostszym przypadku elementy te są umieszczone w wektorze n-elementowym.

Opis algorytmu

Zastosujemy jeden z wielu algorytmów sortowania – sortowanie bąbelkowe. Zasadą tego algorytmu jest porównywanie wartości dwóch sąsiednich elementów i jeżeli są one w relacji odwrotnej względem zadanej, to następuje zamiana miejscami tych elementów. W jednym przejściu całego wektora, metodą kolejnych zamian, element o największej wartości jest przenoszony na koniec wektora. W algorytmie jest wykonywanych tyle podobnych przejść, ile jest elementów (minus 1) w wektorze, przy czym za każdym razem jest przeglądany o jeden element mniej – ten zawierający największą wartość znalezioną w poprzednim przejściu. Działanie algorytmu prześledźmy na rys. 7.13.

Łatwo zauważyć, że algorytm opiera się na dwóch cyklach wyliczanych:

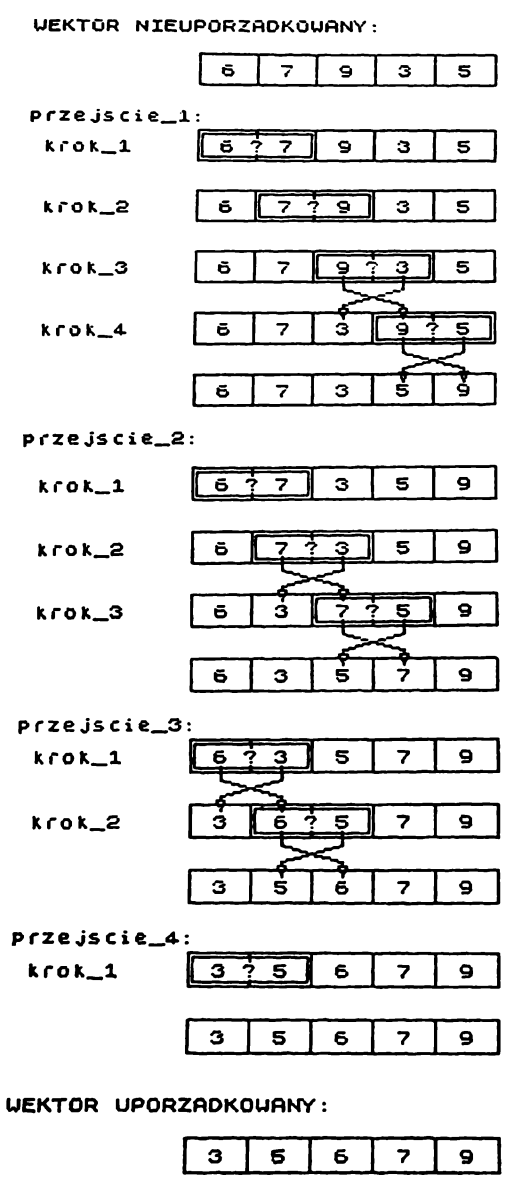
– zewnętrznym od 1 do $n-1$

– wewnętrznym od 1 do n – *zmienna_cyklu_zewnętrznego*

I jeszcze uwaga – może się zdarzyć, że rozpatrywany wektor jest już uporządkowany całkowicie, lub częściowo. Warto więc, dokonując przejścia, sprawdzać czy trzeba było dokonać przestawienia. Jeżeli nie było żadnego przestawienia – wektor jest już uporządkowany.

Analizując złożoność algorytmu rozważmy dwie skrajne sytuacje:

1. Wektor początkowo jest całkowicie uporządkowany. Aby to stwierdzić, program musi dokonać $(n-1)$ sprawdzeń relacji dla par elementów, a więc wykonać $2 * (n-1)$ dostępów do elementów wektora.
2. Wektor jest początkowo całkowicie uporządkowany, ale w kolejności odwrotnej (o czym program z góry nie wie). W trakcie wykonania, program musi dokonać $\sum_{i=1}^{n-1} \sum_{j=1}^{n-i} j = \left(\frac{1+n}{2}\right) * (n-1)$ operacji i w każdej z nich dokonać


WEKTOR NIEUPORZADKOWANY:
6 7 9 3 5
przejscie_1:
krok_1
krok_2
krok_3
krok_4
przejscie_2:
krok_1
krok_2
krok_3
przejscie_3:
krok_1
krok_2
przejscie_4:
krok_1
WEKTOR UPORZADKOWANY:
3 5 6 7 9

Rys. 7.13

sprawdzenia relacji wartości pary elementów oraz dokonać ich wzajemnej wymiany. W najprostszym przypadku jest to 5 dostępów, a przy zoptymalizowaniu wymagane są 4 dostępy – a więc $2*(n+1)*(n-1)$. Przyjmując wartość średnią obu sytuacji skrajnych, średnia liczba dostępów do elementów wektora w tej metodzie wynosi $(n-1)*(n+2)$.

Opis danych

Założymy, że mamy wypełniony n wartościami wektor w (1000) licząc od bazy równej 1.

Opis wyników

Otrzymamy ten sam wektor, ale uporządkowany.

Tekst programu

Jako fragment programu podam zapis algorytmu sortowania na m elementach nieposortowanego wektora w z przykładu 7.1. Jest to więc część D wersji *W2* tego rozwiązania

```
599 '     ------------------------------------
600 REM  Sortowanie wektora z wyprowadzaniem
605 '
610 '    w(100)   - sortowany wektor
615 '    m=w(0)   - aktualne zapelnienie wektora
620 '    p        - zmienna pomocnicza
625 '    f        - wskaznik posortowania
630 '
640 LET m=w(0)
645 FOR i=1 TO m-1
650   LET f=0
655   FOR j=1 TO m-i
660     IF  w(j)<=w(j+1)  THEN  685
665       LET       p=w(j)
670       LET    w(j)=w(j+1)
675       LET w(j+1)=p
680       LET       f=1
685     REM ENDIF
690   NEXT j
695   IF  f=0  THEN  715
700 NEXT i
705 '
710 REM  Wyprowadzanie wektora
715 FOR i=1 TO m
720   PRINT w(i),
725 NEXT i
730 '
735 '  ------------------------------------
```

Styl programowania

Stosowanie tablic. Grupowanie wartości w tablice jest zawsze pożądanym kierunkiem usprawnienia postaci programu oraz bardziej zwartego jego zapisu.

Jednak zapis instrukcji dostępu do elementów tablicy zmniejsza nieco czytelność programu. Warto zauważyć, że przeanalizowanie przebiegu programu dla pojedynczego elementu daje w rezultacie informacje o całym algorytmie. Tablice powinny być więc wykorzystywane w każdym przypadku, w którym te same czynności mają być wykonywane dla wielu zmiennych oraz wtedy, gdy w algorytmie są wykorzystywane relacje między tymi zmiennymi (np. większe, mniejsze, sąsiednie itp.).

Stosowanie tablic ma też znaczenie dla realizacji postulatu modularności programu. W uogólnionym przypadku wykorzystując dwa wektory *dane* (*n*) i *wyniki* (*m*) możemy wyodrębnić trzy moduły programu:

1. Moduł wprowadzania danych do wektora *dane*. Wprowadzane wartości mogą być następnie wstępnie sprawdzone pod kątem ich rzetelności oraz natychmiast wyprowadzone dla weryfikacji przez użytkownika.
2. Moduł obliczeń działający na wartościach z wektora *dane* i pozostawiający wartości w wektorze *wyniki*. Nie jest przy tym istotna logika algorytmu oraz zależność między danymi.
3. Moduł wyprowadzania wyników prezentuje w najprostszej postaci wartości bezpośrednio użytkownikowi. W rozbudowanych wersjach wektor z wynikami może być następnie przetwarzany w celu wyszukania wartości istotnych dla użytkownika. Bardzo pożyteczne jest też często graficzne przedstawienie zmienności wartości wynikowych dla danego zestawu danych.

Stosowanie tablic jest więc godne polecenia. Jednakże przesada jest też szkodliwa, należy więc wybrać rozsądne optimum. Duże tablice zajmują duży obszar pamięci i może jej potem zabraknąć na wyniki pośrednie obliczeń. Warto też pamiętać, że czas dostępu do zmiennej prostej jest znacznie krótszy niż do zmiennej indeksowanej – szczególnie wielowymiarowej.

Efektywne wykorzystywanie tablic. Wielokrotność operacji dla tablic ma podstawowy wpływ na czas działania programu. Stąd też poszukując sposobów skrócenia czasu jego działania, należy głównie analizować możliwość zmniejszenia liczby wykonywanych operacji dla każdego elementu tablicy. Drugim czynnikiem mającym istotny wpływ na czas działania programu jest liczba wykonanych instrukcji cyklicznych. W obu tych przypadkach zysk będzie najbardziej widoczny.

Rozważmy pary przykładów

```
10 FOR i=1 TO n

20   FOR j=1 TO n
30     LET x=x + SQR(i$z)$m(i,j)
40   NEXT j
50 NEXT i
```

```
10 FOR i=1 TO n
15   LET p=SQR(i$z)
20   FOR j=1 TO m
30     LET x=x + p$m(i,j)
40   NEXT j
50 NEXT i
```

Uzyskaliśmy, że wyrażenie SQR (*i***z*) będzie liczone tylko *n* razy zamiast *n***m* razy.

```
10 FOR i=0 TO 5000
20   LET  w(i)=1

30 NEXT i
```

```
10 FOR i=0 TO 5000 STEP 2
20   LET    w(i)=1
25   LET  w(i+1)=1
30 NEXT i
```

Liczba operacji na elementach wektora pozostała bez zmian, ale zmniejszyliśmy o połowę liczbę wykonań instrukcji cyklu. Metoda ta nosi nazwę *rozwijania pętli*

```
10 FOR i=0 TO 5000
20   LET  w(i)=2
30 NEXT i
40 FOR i=0 TO 5000
50   LET  z(i)=0
60 NEXT i
```

```
10 FOR i=0 TO 5000 STEP 2
20   LET   w(i)=2
30   LET w(i+1)=2
40   LET   z(i)=0
50   LET z(i+1)=0
60 NEXT i
```

Podobnie zrealizowaliśmy metodę *ściskania pętli* przez łączenie w jednej instrukcji cyklu operacji dla kilku tablic

```
10 FOR k=0 TO 50
20   IF  a>b  THEN  50
30     LET x(k)=y(k)

40     GOTO  60

50     LET y(k)=x(k)
60 NEXT k
65 ....
```

```
10 IF  a>b  THEN  45
20   FOR k=0 TO 50
30     LET x(k)=y(k)
35   NEXT k
40   GOTO  65
45   FOR k=0 TO 50
50     LET y(k)=x(k)
60   NEXT k
65 ....
```

Wartość wyrażenia relacji a > b jest tutaj niezależna od wykonywania cyklu. Czasem warto zapisać operacje w dwóch różnych instrukcjach cyklu, zamiast w jednej, dla uzyskania przyśpieszenia wykonywania operacji.

Warto zwrócić uwagę, że stosowanie tablic wielowymiarowych jest wskazane tylko wtedy, kiedy ma to uzasadnienie w jasności i przejrzystości algorytmu, a więc wówczas, gdy w obliczeniach dla danego elementu mają znaczenie elementy sąsiednie lub z otoczenia z kilku wymiarów. Czasem też warto dla pewnych przetwarzań tymczasowo umieścić zawartość tablicy wielowymiarowej w wektorze, wykonać obliczenia, po czym wyniki z powrotem wpisać w odpowiednie elementy tablicy.

Sytuacje błędne

Subscript out of range

Indeks poza zakresem. Wartość jednego z indeksów zmiennej indeksowanej przekracza granice określone rozmiarem w danym wymiarze lub liczba indeksów jest niezgodna z liczbą wymiarów podanych w deklaracji tablicy. Jest to

jeden z najczęściej występujących błędów spowodowany pomyłką w zapisie wyrażenia indeksowego lub też błędnym ustaleniem parametrów instrukcji cyklu działającej na elementach tablicy. Drugi przypadek występuje często przy rozwiązywaniu zagadnień wymagających, bądź porównywania ze sobą par elementów tablicy, bądź też przesuwania fragmentu tablicy w inne jej miejsce. Być może nastąpiła też pomyłka w podaniu rozmiaru wierszy lub kolumn, albo przyjęto rozmiar wiersza za rozmiar kolumny lub odwrotnie.

Pojawienie się tego rodzaju błędu może być również wynikiem błędnych danych, od wartości których jest uzależnione obliczenie wartości indeksów. Stąd też należy stosować programowe zabezpieczenia przed powstaniem tego rodzaju błędów – sprawdzając zakresy wartości indeksów przed ich wykorzystaniem w zmiennych indeksowanych. Dla ilustracji rozważmy fragment programu

```
5   DIM  vv(6,8)
10  INPUT i,j,w
20  IF  i<0  THEN 300
22  IF  i>6  THEN 300
24  IF  j<0  THEN 300
26  IF  j>8  THEN 300
30  LET  vv(i,j)=w
35  GOTO 10
40  '
299 '
300 PRINT "Indeks poa zakresem"
305 PRINT "sprawdz i podaj poprawnie"
310 GOTO  10
```

Redimensioned array

Powtórna deklaracja tablicy. W trakcie wykonywania programu rozpoczęto wykonywanie instrukcji deklaracji tablicy, w której występuje nazwa tablicy już wcześniej zadeklarowanej. Identyczna sytuacja wystąpi, gdy przed deklaracją danej tablicy było już odwołanie do jej elementów – nastąpiła wtedy deklaracja standardowa.

W systemie BASIC będącym interpretatorem przyczyną wystąpienia tego błędu może być umieszczenie instrukcji deklaracji tablicy w zakresie instrukcji cyklu. Drugie wykonywanie cyklu spowoduje zatem wystąpienie błędu. Jedynie wcześniejsze wykonanie instrukcji wymazania tablicy umożliwia powtórną deklarację. Pamiętajmy przy tym, że usunięcie tablicy powoduje nieodwołalne wymazanie zapisanych w niej dotychczas wartości. Jeżeli więc chcemy rozszerzyć tablicę już istniejącą, to algorytm postępowania powinien być taki jak to zapisano we fragmencie programu

```
100  LET o$="Stan0"
110  INPUT "Podaj rozmiar wektora "; n
120  DIM  az(n)                  'deklaracja tablicy glownej
130  IF  o$<>"Rozszerz"  THEN  190
140    FOR i=0 TO  nn            'odtworzenie poprzedniej
150      LET  az(i)=pp(i)        ' zawartosci tablicy glownej,
160    NEXT i                    ' elementy  nn+1..n
165                              ' pozostaj puste,
170    LET o$="Stan0"
180    ERASE  pp                 'wymazanie tablicy pomocniczej
190  REM ENDIF
200  ' Wykorzystanie tablicy rozszerzonej
205  '
495  '
500  INPUT "Czy rozszerzyc tablico(Nie/Rozszerz)"; o$
510  IF  o$="Nie"  THEN  590
520    DIM  pp(n)                'deklaracja tablicy pomocniczej,
530    FOR i=0 TO n              'przepisanie do tej tablicy
540      LET pp(i)=az(i)         ' zawartosci tablicy az,
550    NEXT i
560    LET  nn=n                 'zapamietanie starego wymiaru,
570    ERASE  az                 'wymazanie tablicy  az,
580    GOTO  110                 'przejscie do wczytania rozmiaru,
590  REM ENDIF
600  END
```

Problemy

7.1. Podaj liczbę elementów tablic deklarowanych instrukcjami

```
a) OPTION BASE 0
   DIM  w(77)
   DIM  xy(5,7)
   DIM  tt(3,2,4)
   LET  a(3,2)=5
```

```
b) OPTION BASE 1
   DIM  v(1)
   DIM  z(5,7)
   DIM  a4(6,7,8,5)
   LET  z=p(i,j)
```

7.2. Podaj wartości zmiennej *x9* po wykonaniu instrukcji, przy założeniu wypełnienia tablic, jak na rys. 7.14

⟨w⟩	0
0	4
1	2
2	0
3	3
4	1
5	4

⟨m⟩	0	1	2	3	4
0	5	0	4	4	2
1	6	5	2	0	1
2	3	1	3	2	3
3	2	3	1	3	2

Rys. 7.14

```
a)  LET  x9=m(w(2),w(5)) + 1
b)  LET  x9=w(m(1,1)/m(3,4))
c)  LET  x9=w(4) + w(w(w(2)))
d)  LET  x9=m(2,0) * m(0,2)
e)  LET  x9=m(0,w(0)) + m(1,w(1)) + m(2,w(2)) + m(3,w(3))
f)  LET  x9=w(0) + w(1) + w(2) + w(3) + w(4) + w(5)
```

7.3. Dla wielomianu stopnia n ($n<20$) o współczynnikach zapisanych w wektorze $a(n)$, gdzie $a(0)$ jest współczynnikiem wyrazu wolnego, napisać program obliczenia:

a) wartości wielomianu dla danego x, według schematu Hornera:
$(...(((a(n)*x)+a(n-1))*x+a(n-2))*x+...a(1))*x+a(0)$

b) wartość m-tej pochodnej wielomianu dla danego x – współczynniki kolejnej pochodnej wielomianu oblicza się z zależności $b(i-1)=(i-1)*a(i)$ dla $i=1...n$, gdzie $b(n-1)$ jest wektorem pomocniczym,

c) wartości całki wielomianu w przedziale $[a, b]$ – całkę wielomianu w przedziale oblicza się jako różnicę wartości całek dla $x=b$ i $x=a$, a współczynniki całki wielomianu oblicza się z zależności $c(i+1)=a(i)/(i+1)$ dla $i=0...n$, gdzie $c(n+1)$ jest wektorem pomocniczym.

7.4. Napisać program sumowania wartości elementów leżących

a) na głównej przekątnej tablicy $t3(15,15,15)$,

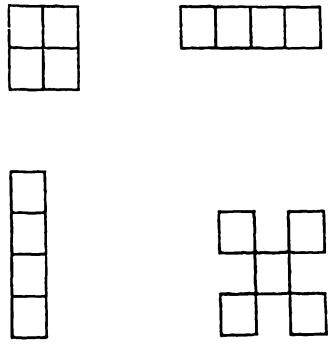
b) poniżej głównej przekątnej tablicy $m2(25,25)$ – bez elementów na niej,

c) w wierszach oraz kolumnach tablicy $kk(33,33)$ zgodnie z zasadami księgowania,

d) jak czarne pola szachownicy tablicy $sz(8,8)$ – (OPTION BASE 1),

e) w tablicy $zz(7,6,5)$ – osobno sumę wartości ujemnych i osobno sumę wartości dodatnich.

7.5. Napisać program znajdowania w tablicy $wi(50,50)$ fragmentu o maksymalnej wartości sumy elementów:



Rys. 7.15

a) przedstawionych na rys. 7.15b,

b) będącego częścią tablicy o dowolnej takiej samej liczbie wierszy i kolumn (należy tutaj założyć, że wartości elementów tablicy mogą być też ujemne),

c) leżących w pojedynczym wierszu i kolumnie,

d) tworzących otoczenie danego elementu.

W wyniku należy zlokalizować położenie i postać fragmentu oraz wyznaczoną sumę.

7.6. W tablicy *al*(50,50) znajdują się liczby naturalne reprezentujące wysokości. Napisz program poruszania się po tablicy figury szachowej tak, aby kolejny jej ruch był zgodny z zasadami jej poruszania oraz aby spadek drogi był minimalny.

Program napisz dla następujących figur:

a) król, b) wieża, c) goniec, d) dama, e) konik

Struktura programu powinna być tak skonstruowana, aby możliwa była prosta wymiana rodzaju figury.

Wskazówka: Tablicę *al*(50,50) ze względu na punkt e) należy włożyć w tablicę *bx*(54,54) wypełnioną w dwóch wierszach i kolumnach brzegowych wartościami równymi 32767.

Let's learn BASIC English

array tablica
vector wektor
matrix macierz
dimension wymiar
size rozmiar (tutaj rozumiany w danym wymiarze)
row wiersz
column kolumna
layer warstwa
subscript, index indeks; wskaźnik (czasem również używane)
range zakres (w domyśle indeksów)
base baza
DIMension WYMiar (właściwie powinno tutaj występować słowo kluczowe deklarujące typ tablicowy – np. **ARRAY** TABLICA)
ERASE WYTRZEĆ; zmazać
OPTION BASE OPCJA BAZY

END of lesson

8. Przetwarzanie tekstów

Komputery zostały skonstruowane przede wszystkim do maksymalnie szybkiego wykonywania obliczeń w zakresie czterech działań arytmetycznych na wartościach liczbowych. Początkowo też porozumiewano się z nimi przez kodowanie rozkazów jako wartości liczbowych. Z czasem uciążliwość kodowania przerzucono na komputery, a rozkazy były zadawane w postaci słów. Rozpoczęła się nowa droga rozwoju komputerów polegająca na rozpoznawaniu i przekształcaniu, czyli przetwarzaniu tekstów.

Przetwarzanie tekstów polega na porównywaniu, zmienianiu, usuwaniu lub dopisywaniu znaków w wartościach typu tekstowego. Komputer praktycznie nie wykorzystuje tutaj swoich umiejętności w zakresie czterech działań. Najczęściej używanymi programami z zakresu przetwarzania tekstów są obecnie programy edycji i redakcji. Pierwszy z nich umożliwia zapisywanie tekstów, w tym tekstów programów, udostępniając też funkcje poprawiania już wprowadzonych informacji. Program redakcji tekstów rozszerza możliwości edycji umożliwiając redagowanie tekstu, czyli jego rozmieszczanie na stronie, prawostronne wyrównywanie, proste zapisywanie tabel oraz wykorzystanie własności drukarek z wieloma graficznymi możliwościami stosowania różnych zestawów znaków.

Obecnie trudno doprawdy stwierdzić, czy większość komputerów osobistych jest zatrudniana do obliczeń numerycznych, czy do przetwarzania tekstów.

8.1. Wyrażenia tekstowe

W rozdziale 4 wprowadzono pojęcie typu tekstowego oraz odpowiednio pojęcie stałej i zmiennej typu tekstowego.

Wartością typu tekstowego są *napisy* będące sekwencją znaków dostępnych w zbiorze znaków systemu BASIC dla danego komputera. W pamięci operacyjnej każdy znak jest przechowywany w postaci wartości liczbowej będącej *kodem znaku*. Przypisanie wartości kodów poszczególnym znakom ze zbioru jest standardowe. Najczęściej spotykanym standardem jest kod ASCII (ISO).

Wykaz wartości kodów ASCII dla zbioru znaków systemu BASIC jest podany w dodatku A. Tutaj warto zauważyć, że litery, osobno duże i małe, są uporządkowane alfabetycznie według wzrastających wartości kodów. Znaki o wartościach kodu mniejszych od 32 są znakami sterującymi i funkcyjnymi. Znaki te, oprócz kilku posiadających odrębne klawisze, są wprowadzane przez wciśnięcie klawisza litery przy wciśniętym klawiszu CTRL. Funkcją klawisza CTRL jest odjęcie od kodu znaku dużej litery wartości 64, dając w rezultacie wartość kodu znaku sterującego.

Przykładowymi wartościami kodów znaków są:

1	CTRL-A	65	A	97	a	48	0
3	CTRL-C	67	C	99	c	51	3
26	CTRL-Z	90	Z	122	z	32	␣spacja

Długość tekstu, określona liczbą znaków, jest atrybutem wartości typu tekstowego. Dopuszczalne jest wykorzystywanie wartości tekstowych złożonych od 0 do 255 znaków. *Wartością pustą* jest tekst bez znaków.

Napis jest zewnętrzną postacią wartości typu tekstowego prezentując wartość tekstową ujętą w cudzysłowy. W niektórych przypadkach możliwe jest zapisywanie napisów bez cudzysłowów.

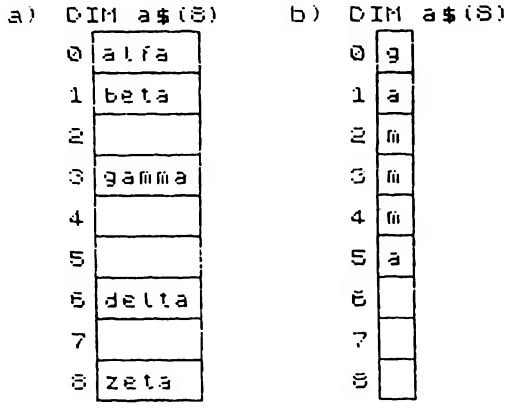
Stałe typu tekstowego są napisami. Zmienne typu tekstowego są identyfikowane dodatkowym symbolem dolara $. Dopuszczalne są też wielowymiarowe tablice typu tekstowego, a więc o elementach przechowujących wartości tekstowe (rys. 8.1a).

Uwaga: W niektórych wersjach języka BASIC, teksty mogą być przechowywane jedynie w zmiennych typu tablicowego o zadeklarowanym rozmiarze, w których każdy element zmiennej przechowuje wartość pojedynczego znaku (rys. 8.1.b).

Przy przetwarzaniu tekstów należy pamiętać, że już cztery zmienne typu tekstowego, przechowujące teksty o maksymalnej długości (255 znaków), zajmują prawie 1K bajtów pamięci operacyjnej.

Instrukcja przypisania umożliwia przypisanie wartości również zmiennej typu tekstowego. Wartość przypisywana musi być również typu tekstowego.

Analogicznie jak dla wyrażeń typu liczbowego określimy pojęcie wyrażenia typu tekstowego.



Rys. 8.1

Wyrażenie typu tekstowego jest budowane z argumentów oraz operatora tekstowego. Argumentem wyrażenia może być stała, zmienna lub odwołanie do funkcji typu tekstowego. *Operatorem tekstowym* jest operator konkatenacji oznaczony symbolem +.

Prostymi wyrażeniami tekstowymi są pojedyncze argumenty
"alfabet" *z$* *a$*(5) LEFT$(*z*,$,3)

Konkatenacją nazywamy złożenie wartości tekstowych z obu argumentów operacji w jedną wartość

```
"mikro" + "komputer" + " ABC"      - wyrazenie tekstowe
"mikrokomputer" + " ABC"           - po wykonaniu 1 operacji
"mikrokomputer"                    - po wykonaniu 2 operacji
```

Wyrażenia tekstowe mogą być wykorzystywane w instrukcjach przypisania i wyprowadzania oraz jako argumenty standardowych funkcji tekstowych.
Dla instrukcji wczytywania oraz wprowadzania wartości zmiennych tekstowych, w strumieniu wejściowym muszą się znajdować napisy. Dopuszczalne jest opuszczenie cudzysłowów w napisie jeżeli tekst nie zawiera znaków przecinka, średnika lub znaczących spacji. Separatorem takich napisów jest bowiem spacja lub przecinek.

Rozważmy program

```
10  READ  a$,b$,c$
20  INPUT x$,y$,z$
30  PRINT a$;x$
40  PRINT b$ + y$
45  LET   m$ = c$ + z$
50  PRINT m$
60  END

100 DATA  Maly, Duzy, "Nie duzy "
RUN
? Stolik, " chlopiec", Wazon
MalyStolik              - brak spacji rozdzielajacej gdyz
                          wczytane zostaly jedynie znaki
                          znaczace.
Duzy chlopiec           - spacja pochodzi z tekstu
                           " chlopiec"
Nie duzy Wazon          - spacja pochodzi z tekstu z zesta-
                          wu danych.
Ok
```

Warto tutaj zauważyć, że w każdym przypadku parametry instrukcji wyprowadzenia zostały podane inaczej:

30 – konkatenacja tekstów nastąpiła w strumieniu wyjściowym,

40 – konkatenacja tekstów nastąpiła przez wykonanie operacji konkatenacji,

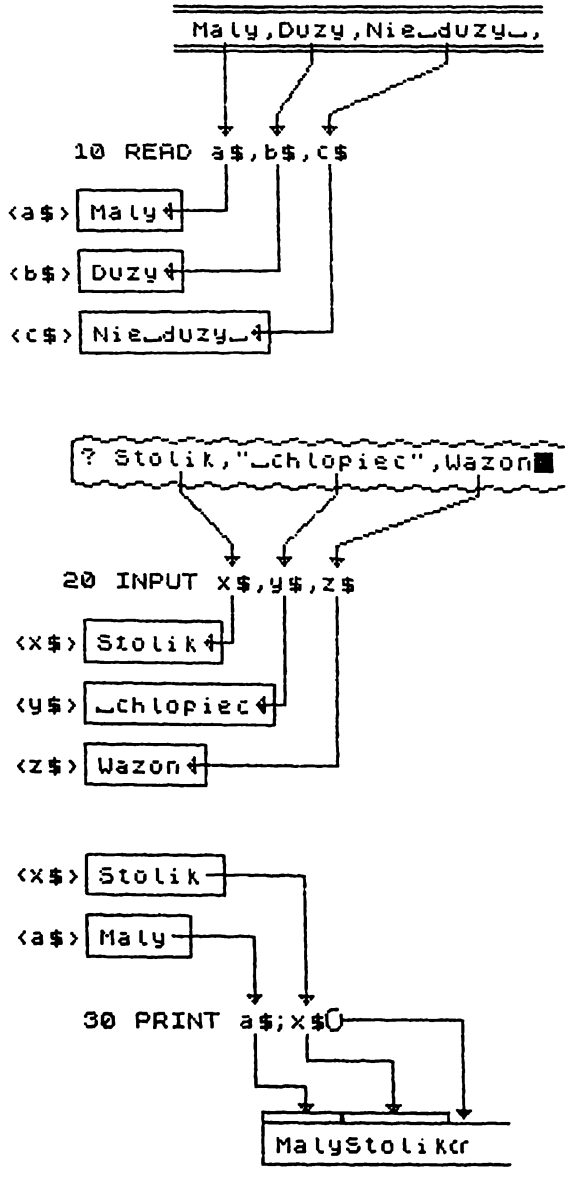
50 – konkatenacja tekstu nastąpiła wcześniej podczas wykonania instrukcji z linii 45.

Wykonanie tego programu jest zilustrowane na rys. 8.2.

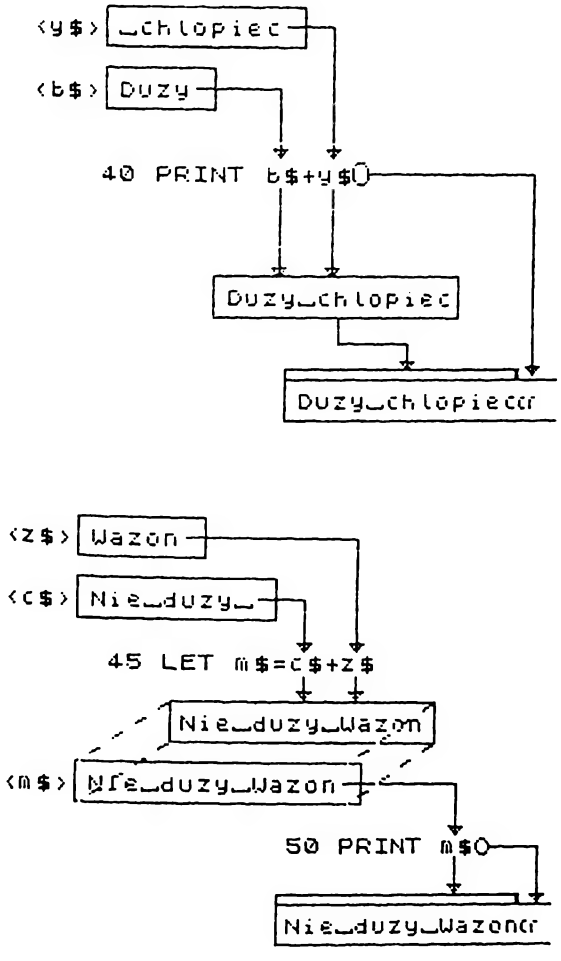
Przy wprowadzaniu dłuższego tekstu jako strumienia danych dla instrukcji wprowadzania, konieczne jest zazwyczaj zawarcie go w cudzysłowach. Zasada ta w niektórych przypadkach nie jest akceptowana przez użytkownika, który nie musi rozumieć niuansów rozpoznawania typu stałej w strumieniu przez system BASIC. Poznajmy więc nieco inną w działaniu instrukcję wprowadzania pełnej linii tekstu.

LINE INPUT *zmienna_tekstowa*
LINE INPUT ; *zmienna_tekstowa*
LINE INPUT "*tekst_zachęty*"; *zmienna_tekstowa*

Instrukcja wprowadza ze strumienia wejściowego danych z klawiatury od 0 do 254 znaków przypisując je pojedynczej *zmiennej_tekstowej*. *Tekst_zachęty* jest wyprowadzany po rozpoczęciu wykonywania instrukcji – znak pytajnika nie jest nigdy wyprowadzany. Znaki cudzysłowów we wprowadzanym napisie są jego częścią. Końcem wprowadzania napisu jest znak RETURN. Średnik występujący bezpośrednio po słowie LINE INPUT oznacza brak zmiany wiersza po zakończeniu wprowadzania napisu.



Rys. 8.2 (część 1)



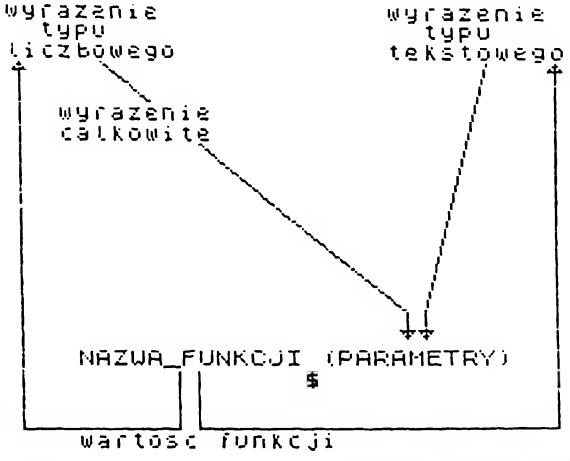
Rys. 8.2 (część 2)

8.2. Standardowe funkcje tekstowe

W algorytmach przetwarzania tekstów można dość szybko wyróżnić kilka podstawowych operacji powtarzających się w różnych zastosowaniach. Do takich najbardziej typowych zaliczyłbym:

a) określenie długości tekstu,
b) wybranie i usunięcie lub zamiana wycinka tekstu,
c) sprawdzenie istnienia w tekście określonego wycinka,
d) utworzenie tekstu z wartości liczbowych.

Większość tych operacji w systemie BASIC jest realizowana za pomocą standardowo wbudowanych funkcji tekstowych. Funkcje te mogą mieć od 1 do 3 argumentów, w tym typu liczbowego lub tekstowego. Również wśród funkcji tekstowych znajdują się funkcje o wartościach typu liczbowego. Zależności te ilustruje rys. 8.3.



Rys. 8.3

Wywołanie standardowej funkcji tekstowej

nazwa_funkcji (*argumenty*)

Nazwa_funkcji jest identyfikatorem funkcji. *Nazwa_funkcji* o wartości tekstowej zawiera po identyfikatorze symbol dolara $. *Argumenty* są listą argumentów oddzielonych przecinkami, będących poprawnie skonstruowanymi wyrażeniami typu liczbowego bądź tekstowego

Dla uproszczenia zapisu funkcji przyjmiemy dalej, że
I, J – oznacza argument będący wyrażeniem typu całkowitego,
X$, Y$ – oznacza argument będący wyrażeniem typu tekstowego.

Tekstowe funkcje standardowe o wartościach liczbowych:

ASC (X$)	– wartość kodu pierwszego znaku tekstu,
LEN (X$)	– liczba znaków w tekście,
VAL (X$)	– wartość liczbowa równoważna tekstowi reprezentującemu zapis liczby,
INSTR (X$, Y$)	– sprawdzenie istnienia tekstu podanego argumentem,
INSTR (I, X$, Y$)	– Y$ w tekście podanym argumentem Y$.

Funkcja ASC(X$) udostępnia wartość kodu pierwszego znaku tekstu. Możemy więc na przykład szybko rozpoznać tekst zapisując

```
10  PRINT "Czy gramy dalej (TAK/NIE)";
20  INPUT  a$
30  LET  i=ASC(a$)
40  IF  i=78  THEN  END          ' 78  - kod litery  N
50  IF  i=110 THEN  END          '110  - kod litery  n
60  PRINT "Ok - gramy dalej"
```

Funkcja LEN(X$) podaje liczbę znaków tekstu będącego wartością argumentu. Dla tekstu pustego wartością funkcji jest zero.

```
PRINT  LEN("");LEN("BASIC");LEN("128")
 0  5  3
Ok
```

W trzecim wywołaniu funkcji argumentem jest tekst (a nie liczba!), a więc wynik podaje liczbę cyfr w tej „liczbie".

Funkcja VAL(X$) pozwala na przetworzenie sekwencji znaków tekstu tworzącej zapis liczby na równoważną jej wartość liczbową. Spacje poprzedzające znaki liczby są pomijane. Porównajmy więc dwa fragmenty programów, równoważnych funkcjonalnie

```
10  INPUT  x
20  PRINT  x+1
RUN
? 8
 9
Ok
```

```
10  INPUT  x$
20  PRINT  VAL(x$)+1
RUN
? 8
 9
Ok
```

Przy okazji proszę się zastanowić, jak można rozróżnić te programy, nie widząc ich tekstu — korzystając tylko z możliwości ich wielokrotnego wykonywania.

Omówienia funkcji INSTR dokonamy dalej, po zaprezentowaniu drugiej grupy standardowych funkcji tekstowych.

Tekstowe funkcje standardowe o wartościach tekstowych:

CHR$(I)	— utworzenie jednoznakowego tekstu,
LEFT$(X$,I)	— wybranie wycinka z lewej strony tekstu,
MID$ (X$,I) MID$ (X$,I,J)	— wybranie wycinka ze środka tekstu,
RIGHT$ (X$,I)	— wybranie wycinka z prawej strony tekstu,
SPACE$ (I)	— utworzenie tekstu złożonego z I spacji,
STRING$ (I,J) STRING$ (I,X$)	— utworzenie tekstu złożonego z I jednakowych znaków,
STR$ (X)	— utworzenie tekstu reprezentującego wartość X.

Funkcja CHRS(I) jest przede wszystkim wykorzystywana do tworzenia tekstów złożonych ze znaków sterujących. Znaki te służą do sterowania działaniem urządzeń. Przykładowo dla ekranu znaki o podanych wartościach kodu powodują:

7 – wygenerowanie sygnału dzwonka (buczka),
8 – przesunięcie kursora w lewo z wymazaniem znaku (CTRL-H),
10 – przesunięcie kursora na początek wiersza w dół (LINEFEED),
13 – przesunięcie kursora na początek nowej linii (RETURN, ENTER),
22 – przesunięcie całej zawartości ekranu o jeden wiersz w dół,
23 – przesunięcie całej zawartości ekranu o jeden wiersz w górę,
25 – sprowadzenie kursora w lewy górny punkt ekranu,
26 – wymazanie zawartości linii z kursorem,
28 – przesunięcie kursora w prawo.

Niestety, prawie dla każdego monitora z ekranem dużej części kodów są przypisywane nieco inne funkcje. Jest to jeden z większych problemów przy przenoszeniu programów z jednego typu komputera na inny – jeżeli programy te korzystają z tego rodzaju znaków sterujących

```
PRINT  CHR$(7);CHR$(34);"NAPIS";CHR$(34)
"NAPIS"
Ok
```

Zlecenie to spowodowało dźwiękowe zasygnalizowanie wyprowadzenia tekstu NAPIS w cudzysłowach (?). Innej możliwości wyprowadzenia znaku cudzysłowu nie ma, gdyż jest on traktowany jako separator napisu. Dla porządku zapiszmy równoważność

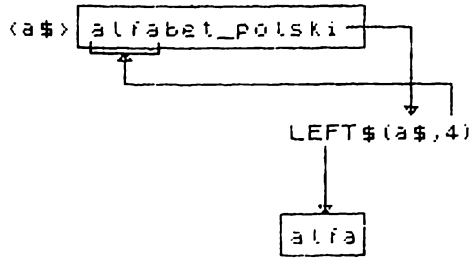
a) CHR$(ASC(X$)) ≡X$, jeżeli LEN(X$) = 1
b) ASC(CHR$(I)) ≡ I, jeżeli I = 0..127

Funkcje LEFT$, MID$, RIGHT$ pozwalają na wybranie z tekstu określonego fragmentu (wycinka). Wszystkie znaki w tekście mają określony numer pozycji, począwszy od wartości 1 do LEN(X$). Wartość równa zero oznacza tekst pusty.

Wartością funkcji LEFT$(X$,I) jest tekst złożony z pierwszych I znaków tekstu X$ (rys. 8.4).

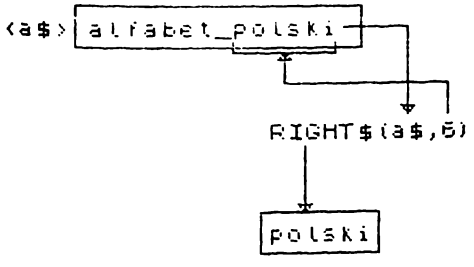
Wartością funkcji RIGHT$(X$,I) jest tekst złożony z ostatnich I znaków tekstu X$ (rys. 8.5).

Funkcja MID$(X$,I) wybiera z tekstu X$, znak z I-tej pozycji. Funkcja MID$(X$,I,J) wybiera zaś z tekstu X$, J znaków począwszy od I-tej pozycji (rys. 8.6).



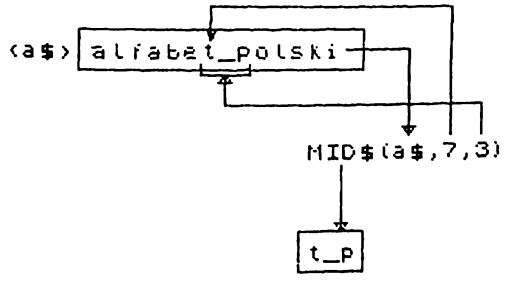

```
LEN(LEFT$(a$,4))≡4
```

Rys. 8.4




```
LEN(RIGHT$(a$,6))≡6
```

Rys. 8.5




```
LEN(MID$(a$,7,3))≡3
```

Rys. 8.6

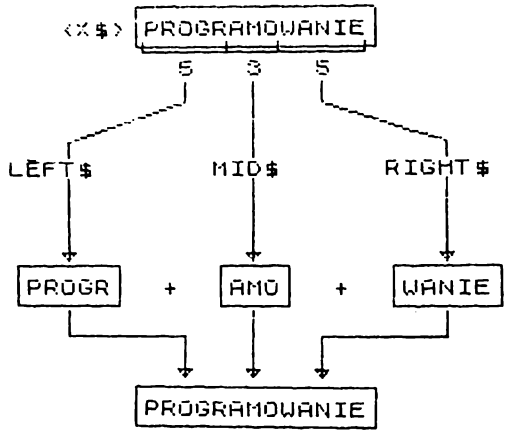
Zanotujmy tutaj następujące równoważności, zakładając że wartość I należy do przedziału 1..LEN(X$):

a) LEFT$(X$,I) ≡ MID$(X$,1,I)

b) RIGHT$(X$,I) ≡ MID$(X$,LEN(X$)−I,J)

c) LEFT$(X$,I) + MID$(X$,I + 1,J) + RIGHT$(X$,LEN(X$)−I−J) ≡ X$
jeżeli J < LEN(X$)−I

Równoważność (c) jest pokazana na rys. 8.7.



Rys. 8.7

Dla ilustracji wykorzystania podanych funkcji, rozpatrzmy fragment tekstu programu oraz wynik jego wykonania.

```
200  INPUT  t$
210  FOR  i=1  TO  LEN(t$)
220    PRINT  RIGHT$(t$,LEN(t$)-i);LEFT$(t$,i)
230  NEXT i
RUN
? ALFABET
LFABETA
FABETAL
ABETALF
BETALFA
ETALFAB
TALFABE
ALFABET
Ok
```

Wydaje się, że komentarz jest tutaj zbyteczny.

Funkcja SPACE$(I) powoduje wytworzenie tekstu złożonego z I spacji. Funkcję tę stosujemy głównie przy organizacji postaci wyświetlanych lub drukowanych wyników działania programu. Znajduje też zastosowanie, jeżeli chcemy dopełnić istniejący tekst do określonej długości. Dla przykładu tekst *k*$ chcemy wyświetlić począwszy od 5 pozycji w wierszu oraz na końcu tego wiersza chcemy umieścić znak gwiazdki. Załóżymy w tym przypadku, że wiersz ekranu składa się z 60 znaków.

```
PRINT  SPACE$(4);k$;SPACE$(60 - LEN(k$) -5);"*"
```

Funkcja STRING$(I,J) oraz STRING$(I,X$) powoduje wytworzenie tekstu złożonego z I znaków o wartości kodu równym J lub wartości kodu pierwszego znaku tekstu X$. Możemy więc zapisać równoważności:

a) STRING$(I, 32) ≡ SPACE$(I)
b) STRING$(I, X$) ≡ STRING$ (I, ASC(X$))

Funkcja STR$(X$) jest funkcją odwrotną do funkcji VAL$(X$) tworzącą tekst reprezentujący wartość typu liczbowego. Funkcja ta jest wbudowana też w wykonywanie instrukcji wyprowadzenia PRINT. Prawdziwe są więc równoważności:

a) VAL(STR$(X)) ≡ X
b) PRINT X ≡ PRINT STR$(X)

ale nie zawsze jest prawdziwe

STR$(VAL(X$)) = X$

(pytanie – dlaczego?).

Funkcja INSTR(I, X$, Y$) lub INSTR(X$, Y$) jest przeznaczona do wyszukiwania w danym tekście X$ wycinka tekstu Y$. Parametr I określa numer pozycji w tekście X$, od którego należy rozpocząć poszukiwanie wycinka. Brak parametru I oznacza numer pozycji równy 1.

Wynikiem działania funkcji jest wartość typu liczbowego podająca:

1..LEN(X$) – numer pozycji w tekście X$, od której rozpoczyna się tekst poszukiwanego wycinka Y$
0 – brak w tekście X$ poszukiwanego wycinka lub początkowo I > LEN(X$) lub tekst X$ jest pusty (LEN(X$) = 0)
I lub 1 – poszukiwany wycinek Y$ jest pusty (LEN(Y$) = 0)

Działanie tej funkcji rozpatrzmy na prostym przykładzie programu:

```
10  LET  p$="Programowanie w jezyku BASIC"
20  LET  t$="BASIC dla poetow"
30  LET  i$="BASIC - Czy to jest Basic"
40  LET  w$="BASIC"
50  PRINT  INSTR(1,p$,w$);INSTR(t$,w$);INSTR(10,i$,w$)
60  END
RUN
 24   1   0
Ok
```

W tekście *p*$ znaleziono poszukiwany wycinek począwszy od 24 pozycji. Podobnie w tekście *t*$, już od pierwszej pozycji. W trzecim tekście pierwszy parametr równy 10 określa zakres poszukiwań od 10 pozycji, a ponieważ wycinek "BASIC" nie jest identyczny z fragmentem tekstu "Basic" – otrzymaliśmy trzeci wynik równy zero.

Działanie funkcji INSTR zapiszemy w postaci równoważnego programu, wykorzystując pozostałe funkcje tekstowe.

```
100  REM  INSTR( i, x$, y$ )
105  '
110  ' Argumenty          : i, x$, y$
115  ' Wynik funkcji      : in
120  ' zmienne lokalne    : i9, j9
125  '
130  LET in=0
135  IF  LEN(x$)=0  THEN  GOTO  205
140  IF  i>LEN(x$)  THEN  GOTO  205
145  IF  i<1        THEN  GOTO  205
150  LET in=i
155  IF  LEN(y$)=0  THEN  GOTO  205
160  FOR i9=i TO  LEN(x$)+1-LEN(y$)
165    IF  ASC( MID$(x$,i9) ) <> ASC(y$)  THEN  195
170    FOR j9=1 TO LEN(y$)
175      IF ASC(MID$(x$,i9-1+j9)) <> ASC(MID$(y$,j9)) THEN 195
180    NEXT j9
185    LET in=i9
190    GOTO  205
195  NEXT i9
200  LET in=0
205  REM  KONIEC INSTR
```

Jako komentarz należy dodać, że podane tutaj rozwiązanie nie jest najbardziej optymalne.

Opisane funkcje tekstowe pozwalają na dowolne przetwarzanie tekstu zgodnie z podanymi zasadami. Z przedstawionych na początku tego rozdziału funkcji ważna w zastosowaniach jest możliwość wymiany określonego fragmentu tekstu. Dla przykładu skonstruujemy fragment programu wymiany, we wprowadzanych z klawiatury tytułach książek, terminu "BASIC" na termin "Fortran".

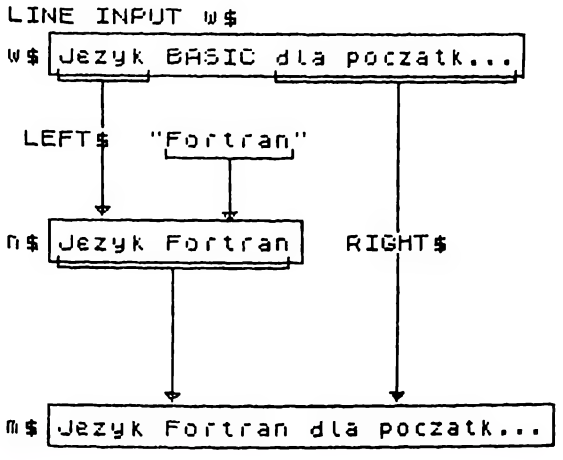
```
10  LINE INPUT w$
20  IF  ASC(w$)=ASC("@")  THEN  END
30  LET k=INSTR(w$,"BASIC")
40  IF  k=0  THEN  100
50    REM  Wymiana wycinka  k..k+LEN("BASIC")-1
60    LET n$=LEFT$(w$,k-1) + "Fortran"
70    LET m$=n$ + RIGHT$(w$,LEN(w$)-k-LEN("BASIC")+1)
80    PRINT  m$
90    GOTO  10
100 PRINT "Brak poszukiwanego slowa"
110 GOTO 10
```

Sposób realizacji wymiany wycinka tekstu zapisanego w programie w linii 60 i 70 przedstawia schematycznie rys. 8.8.

Wymiana w tekście X$ wycinka tekstu o długości J od pozycji I na tekst Y$ może być w ogólnym przypadku wykonana z wykorzystaniem funkcji tekstowych LEFT$ i RIGHT$

$$\text{LEFT\$(X\$,I}-1)+\text{Y\$}+\text{RIGHT\$(X\$, LEN(X\$)}-\text{I}-\text{J}-1))$$

W niektórych implementacjach języka BASIC spotyka się instrukcję przypisania w postaci funkcji – zwaną pseudofunkcją wymiany tekstu.



Rys. 8.8

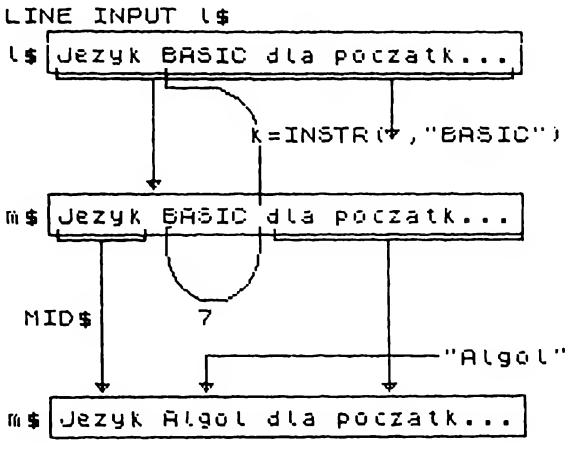
MID$(*zmienna_tekstowa, pozycja*) = *wyrażenie_tekstowe*
MID$(*zmienna_tekstowa, pozycja, liczba_znaków*)
= *wyrażenie_tekstowe*

Instrukcja pseudofunkcji wymienia w tekście wskazanym *zmienną_tekstową* – począwszy od podanej *pozycji* – wycinek tekstu na tekst podany *wyrażeniem_tekstowym. Liczba_znaków* wymienionych w tekście jest podana w parametrach lub jest równa długości *wyrażenia_tekstowego* – przy czym nie może ona przekraczać długości (granicy) tekstu modyfikowanego.

Dla ilustracji przeanalizujmy fragment programu o funkcjach podobnych do poprzedniego.

```
10  LINE INPUT w$
20  IF  ASC(w$)=ASC("$")  THEN  END
30  LET k=INSTR(w$,"BASIC")
40  IF  k=0  THEN  100
50    REM  Wymiana wycinka "BASIC" na "Algol"
60    LET n$=w$
70    MID$(n$,k)="Algol"
80    PRINT n$
90    GOTO  10
100 PRINT "Brak poszukiwanego slowa"
110 GOTO 10
```

Istotną różnicą jest ograniczenie wymiany tylko do tekstów równej długości (rys. 8.9).



Rys. 8.9

8.3. Wyrażenia relacji

Z porównywaniem tekstów spotykamy się przy wyszukiwaniu haseł w słowniku lub encyklopedii. Również większość list porządkujemy według kolejności alfabetycznej liter. W języku BASIC istnieje również możliwość porównywania tekstów przy wykorzystaniu znanych już nam operatorów relacji.

Porównywanie tekstów.

1. Dwa teksty są identyczne (równe), jeżeli zawierają taką samą liczbę znaków oraz kolejno parami wszystkie znaki są identyczne. Dokładniej mówiąc – identyczne muszą być wartości kodów tych znaków.
2. Przy porównywaniu pary tekstów jeden jest większy od drugiego jeżeli:
   a) w przypadku tekstów równej długości przy kolejnym, od lewej strony, porównywaniu par znaków z obu tekstów zostanie wykryta pierwsza wartość kodu znaku większa od wartości kodu drugiego znaku z pary,
   b) w przypadku tekstów różnej długości – tekst dłuższy zawiera w sobie od lewej strony tekst krótszy lub wykryto w nim znak o wartości kodu większej od wartości kodu znaku z odpowiedniej pary znaków.
3. Wynikiem porównywania tekstów jest wartość logiczna *prawda* (zapisywana liczbowo jako –1) lub *fałsz* (liczbowo 0) zależnie od relacji między tekstami.

Przy porównywaniu tekstów brane są pod uwagę wartości kodów znaków, z których są one złożone. Dla standardu ASCII, wartości kodów liter dużych i osobno małych są uporządkowane alfabetycznie (patrz dodatek A).

Wykonanie podanych wyrażeń relacji daje wartości

"BASIC" = "BASIC"	(–1)	prawda; równa długość i identyczne znaki;
"ALGOL" < "BASIC"	(–1)	prawda; równa długość oraz ASC("A") < ASC("B") [66 < 65];
"BASIA" < "BASIC"	(–1)	prawda; ASC("A") < ASC("C") (rys. 8.10a);
"BASIC" = "Basic"	(0)	fałsz; ASC("A") < > ASC("a") [65 < > 97];
"Fortran" < "Fortranowy"	(–1)	prawda; istnieje prawy wycinek dłuższego tekstu „owy" (rys. 8.10b)

Powstaje tutaj pytanie, w jaki sposób porównywać teksty złożone z dużych i małych liter alfabetu – na przykład dla potrzeb układania słownika. Niestety, przed porównywaniem, oba teksty należy przekształcić do tej samej wielkości liter. Tutaj podamy przekształcenie w duże litery.

```
10  REM  UPPER$ - przeksztalcanie w duze litery
15  '
20  ' Tekst wejsciowy  : t$
25  ' Tekst wyjsciowy  : n$
30  '
35  LET  n$=""
40  FOR  i=1  TO  LEN(t$)
45    LET p$=MID$(t$,i,1)
50    IF  p$>"Z"  THEN  p$=CHR$( ASC(p$)-32 )
55    LET n$=n$ + p$
60  NEXT i
65  REM KONIEC UPPER$
```

Oprócz porównywania tekstów złożonych z liter można porównywać też teksty zawierające pozostałe znaki

"" = ""	(–1)	prawda; dwa teksty puste są identyczne;
"121" < "221"	(–1)	prawda; ASC("1") < ASC("2"), ale
"999" < "1000"	(0)	fałsz, gdyż ASC("9") > ASC("1");
"<" < " = "	(–1)	prawda, gdyż ASC(" < ") < ASC(" = ").

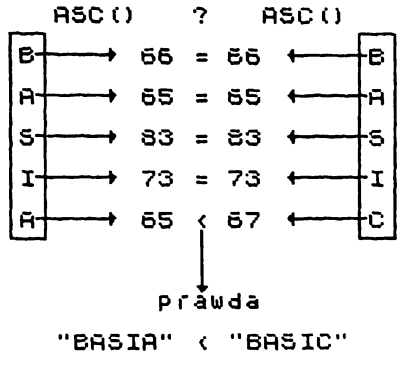
Teraz możemy już rozszerzyć definicję wyrażenia relacji z rozdz. 6.1.

Wyrażenie relacji porównuje wartości dwóch wyrażeń typu liczbowego lub tekstowego

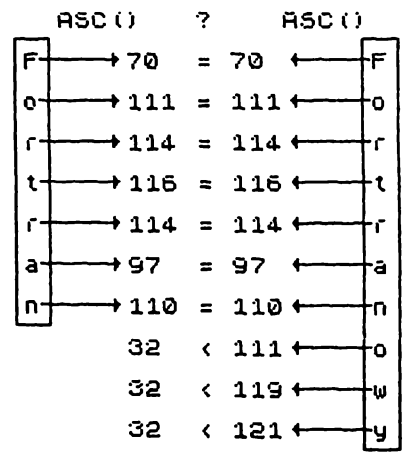
wyrażenie operator_relacji wyrażenie

Operator_relacji określa rodzaj porównania. Oba argumenty relacji muszą być tego samego typu.

a)



b)



Rys. 8.10

Uzupełniając podane przykłady wyrażeń relacji dla tekstów należy przypomnieć, że wyrażeniem tekstowym może być też stała, zmienna lub funkcja typu tekstowego.

LEFT$(*k*$,i) < LEFT$(*k*$,i + 1)	(−1) prawda dla i < LEN(*k*$) − 1;
LEFT$(*k*$,i) = MID$(*k*$,1,i)	(−1) prawda dla i ⩽ LEN(*k*$);
k$ = *k*$	(−1) prawda;
k$ < = *k*$ + *t*$	(−1) prawda, równe gdy *t*$ = "".

IF *wyrażenie_relacji* **THEN** *numer_linii*

Instrukcja warunkowa na tych samych zasadach wykorzystuje wartość wyrażenia relacji dla wartości tekstowych jak dla wartości liczbowych

Typowym, najprostszym zastosowaniem instrukcji warunkowej jest sprawdzenie odpowiedzi użytkownika programu na zadane mu z programu pytanie. Rozważmy fragment programu

```
10  PRINT "Wybierz rodzaj funkcji programu"
20  PRINT "(Powtorz,Wydrukuj,Sortuj,Zakoncz)"
30  INPUT c$
40  IF  c$="Powtorz"   THEN  1000
50  IF  c$="Wydrukuj"  THEN  2000
60  IF  c$="Sortuj"    THEN  3000
70  IF  c$="Zakoncz"   THEN  4000
```

Rozwiązanie to nie jest najlepsze, bo użytkownik może zapisać ten tekst dużymi lub małymi literami oraz może się pomylić we wprowadzaniu tych dość długich poleceń. Lepsze jest rozwiązanie rozpoznawania polecenia po rozpoznaniu pierwszego znaku.

```
10  PRINT "Wybierz rodzaj funkcji programu"
20  PRINT "(Powtorz,Wydrukuj,Sortuj,Zakoncz)"
30  INPUT c$
35  LET  c$=left(c$,1)
40  IF  c$="P"   THEN  1000
45  IF  c$="p"   THEN  1000
50  IF  c$="W"   THEN  2000
55  IF  c$="w"   THEN  2000
60  IF  c$="S"   THEN  3000
65  IF  c$="s"   THEN  3000
70  IF  c$="Z"   THEN  4000
75  IF  c$="z"   THEN  4000
```

Rozwiązanie to możemy uzupełnić wyprowadzeniem pełnego identyfikatora polecenia z oczekiwaniem na wprowadzenie przez użytkownika potwierdzenia – T/N lub t/n (czyli tak/nie).

Drugie zastosowanie jest związane ze wspomnianym już wyszukiwaniem lub porządkowaniem tekstów. Sam algorytm wyszukiwania jest podobny do przykładu 7.1. Również algorytm sortowania może być podobny, z tym, że wielokrotne przestawienie wartości tekstowych, które mogą być złożone z 255 znaków, jest dość kosztowne. Przypominam tutaj problemy z porównywaniem tekstów złożonych z małych i dużych liter. Obecnie dostępne komputery osobiste mają też problemy z akceptacją i porządkowaniem tekstów zawierających charakterystyczne polskie litery – ą, ę, ć itp. (a właściwie to ich nie mają – bo znaków tych brakuje na klawiaturze).

8.4. Przykład konstruowania programu

W punkcie tym zaprezentuję rozwiązanie jednego zadania, opisując kolejne fazy poszukiwania właściwego algorytmu. Opis ten powstawał na bieżąco, gdy przygotowywałem to rozwiązanie dla potrzeb tej książki. Myślę, że dokładne przestudiowanie tego przykładu pozwoli Czytelnikowi poznać metody pracy programisty poszukującego prawidłowego rozwiązania. W drugiej części opisu zamieściłem też omówienie historii uruchomiania i weryfikacji programu na zaprzyjaźnionym komputerze osobistym.

Przykład 8.1. "Analiza tekstu"

Opis problemu

Dla tekstu wprowadzanego z klawiatury, złożonego z sekwencji zdań należy wyznaczyć średnią długość słowa oraz średnią długość zdania.

Opis poszukiwania rozwiązania

W konstruowaniu programu, a więc również w poszukiwaniu algorytmu, będziemy korzystać z możliwości dostarczanych w języku BASIC instrukcji i funkcji operujących na tekstach. Na początku założymy, że kolejne wiersze tekstu będziemy wprowadzać instrukcją LINE INPUT /$, czyli pod zmienną tekstową /$. Z opisu tej instrukcji wiemy, że pojedyncze jej wykonanie umożliwia wprowadzenie tekstu o długości od 0 do 254 znaków. Założymy również na początku, że:

1. Wprowadzone teksty puste będziemy pomijać, a więc

   IF LEN(/$) = 0 THEN *wprowadź_następny*

2. W każdej wprowadzonej linii może być część zdania, pojedyncze zdanie lub kilka zdań – każde zdanie jest kończone znakiem kropki.
3. Większość realizacji instrukcji LINE INPUT nie dopuszcza wprowadzenia więcej niż 254 znaków, co jest sygnalizowane z reguły buczeniem oraz blokowaniem wprowadzania kolejnych znaków. Tutaj dodatkowo poprosimy użytkownika, aby ostatnie słowo w linii (ewentualnie zakończone znakiem kropki jeżeli jest ostatnim w zdaniu) nie było kontynuowane w następnej linii. Tak podzielone słowo będziemy traktować jako dwa słowa.
4. Zakończenie wprowadzenia tekstu do analizy będzie sygnalizowane wprowadzeniem znaku krzyżyka na początku wprowadzania nowej linii, a więc możemy sobie już zaprojektować instrukcję

   IF ASC(/$) = ASC("#") THEN *wyprowadzenie_wyniku.*

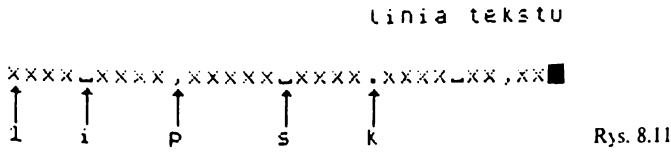
Obecnie zajmiemy się poszukiwaniem realizacji podstawowej części algorytmu. Na początek założymy, że:

ls – licznik słów,
lz – licznik zdań,
ll – licznik wprowadzonych linii,
sd – suma długości słów.

Początkowe wartości tych zmiennych są równe zeru. Po zakończeniu wprowadzenia wszystkich linii tekstu uzyskujemy wyniki:

ls – liczba wczytanych słów,
lz – liczba wczytanych zdań,
ll – liczba wczytanych linii (dla kontroli),
sd/ls – średnia długość słowa,
sd/lz – średnia długość zdania (w długości zdania uwzględniamy tylko długość słów).

Dalej przyjmiemy, że separatorem słowa w tekście może być spacja, kropka lub przecinek, a separatorem zdania – kropka. Teraz pomoże nam rys. 8.11.



Rys. 8.11

Korzystając z funkcji INSTR, szukamy we wprowadzonym tekście spacji, przecinka i kropki, począwszy od pozycji wskazanej zmienną i (początkowo dla każdej wprowadzonej linii $i = 1$)

```
s=INSTR( i, l$, " " )          - nastepna pozycja spacji
p=INSTR( i, l$, "," )          - nastepna pozycja przecinka
k=INSTR( i, l$, "." )          - nastepna pozycja kropki
```

Zaczniemy od $s <> 0$, czyli istnieje następna spacja (w zapisie fragmentów programu na razie numery linii pomijamy)

```
REM   sprawdzanie_spacji
IF  s=0  THEN  wczytanie_nastepnej_linii
  REM  zapisanie_nowego_slowa
    ls=ls+1                  'kolejne slowo zostalo wykryte
    sd=sd+s-i                'dodanie dlugosci kolejnego slowa
     i=s+1                   'przesuniecie pozycji analizowane-
                             ' go tekstu
REM  KONIEC  sprawdzenia_nowego_slowa
```

Podobnie możemy zapisać fragmenty programu jeżeli znaleziono przecinek oraz kropkę. Dla wykrytego znaku kropki musimy dodatkowo doliczyć nowe zdanie $lz = lz + 1$.

Teraz możemy połączyć te fragmenty pamiętając, że separatorem kolejnego słowa może być jeden z trzech znaków oraz, że któryś lub nawet wszystkie

mogą już w analizowanym wycinku tekstu nie występować. Możemy więc mieć następujące wyniki z wykonania funkcji INSTR:

$s=0$ – brak następnej spacji w linii,
$p=0$ – brak następnego przecinka w linii,
$k=0$ – brak następnej kropki w linii,
jeżeli $s<>0$, $p<>0$, $k<>0$ to wtedy, jeżeli

$s<p<k$ lub $s<k<p$ – najbliższym separatorem jest znak spacji,
$p<s<k$ lub $p<k<s$ – najbliższym separatorem jest znak przecinka,
$k<s<p$ lub $k<p<s$ – najbliższym separatorem jest znak kropki,

oraz przypadki pośrednie kiedy któregoś ze znaków już nie ma w pozostałej części tekstu. Trochę jest to skomplikowane, a więc spróbujmy uporządkować te zależności korzystając z tablicy decyzyjnej (jest to sformalizowana metoda weryfikacji poprawnego uwzględniania wszystkich warunków – tutaj korzystamy z wersji bardzo uproszczonej) przedstawionej na rys. 8.12. W górnej części tablicy wypisano wszystkie możliwe kombinacje warunków określających występowanie charakterystycznych znaków w tekście. W dolnej części zaznaczono, jaką czynność należy wykonać, jeżeli dany iloczyn warunków w kolumnie zostanie spełniony.

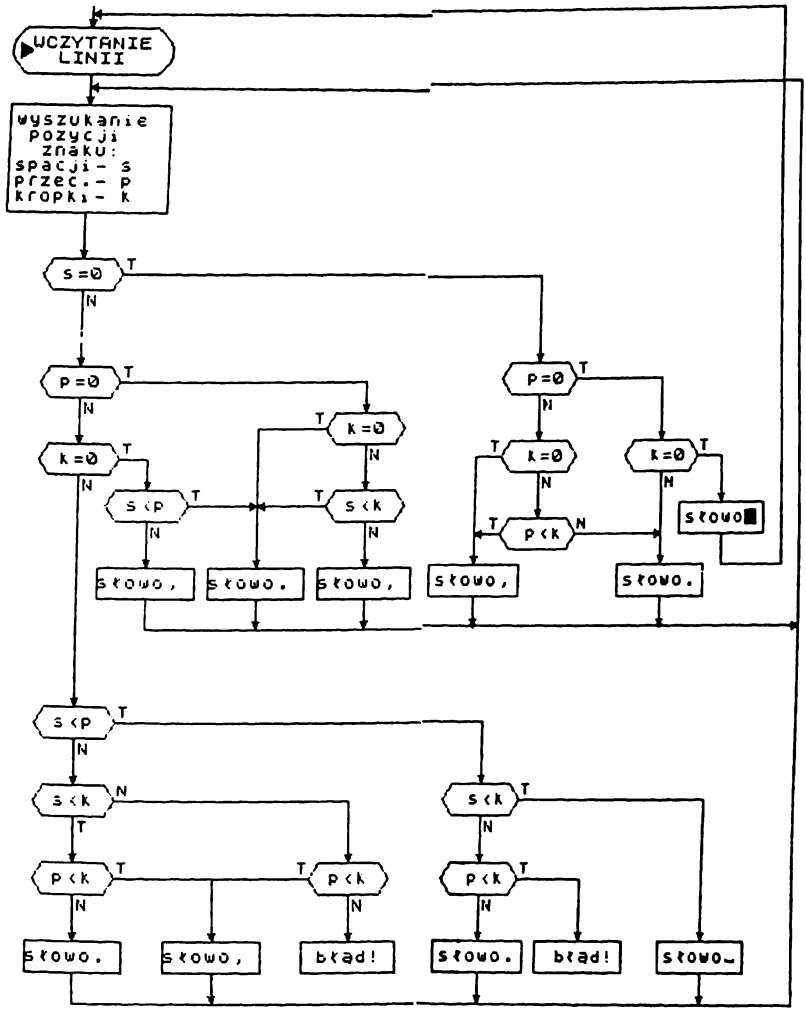
	1	2	3	4	5	6	7	8	10		12		14		15		18	
s=0	T	N	T	T	N	N	N	N	T	T	N	N	N	N	N	N	N	N
p=0	T	T	N	T	N	N	T	T	N	N	N	N	N	N	N	N	N	N
k=0	T	T	T	N	T	T	N	N	N	N	N	N	N	N	N	N	N	N
s<p					T	N					T	T	T	T	N	N	N	N
s<k							T	N			T	T	N	N	T	T	N	N
p<k									T	N	T	N	T	N	T	N	T	N
słowo␣		V			V		V				V	V						
słowo,			V			V			V						V		V	
słowo.				V				V		V				V				V
słowo■	V																	
													błąd					

Rys. 8.12

Przy wypisywaniu warunków wykryto przy okazji dwie kombinacje warunków, których spełnienie oznacza błąd logiczny algorytmu. Na podstawie tej tablicy można stosunkowo łatwo uzyskać sieć działań (rys. 8.13), w której kolejne drogi sterowania są opisane numerami kombinacji.

Otrzymaliśmy dość skomplikowaną sieć działań, której zapisanie w języku programowania będzie również dość skomplikowane.

Pomyślmy więc, czy nie można uprościć naszego algorytmu zmieniając nieco jego koncepcję. Z sieci działań widzimy, że duża liczba warunków dotyczy sprawdzenia, czy wynik funkcji INSTR nie jest równy zeru oraz co gorsze nie można sprawdzać warunków kolejności wystąpienia znaków, jeżeli jednego z nich



Rys. 8.13

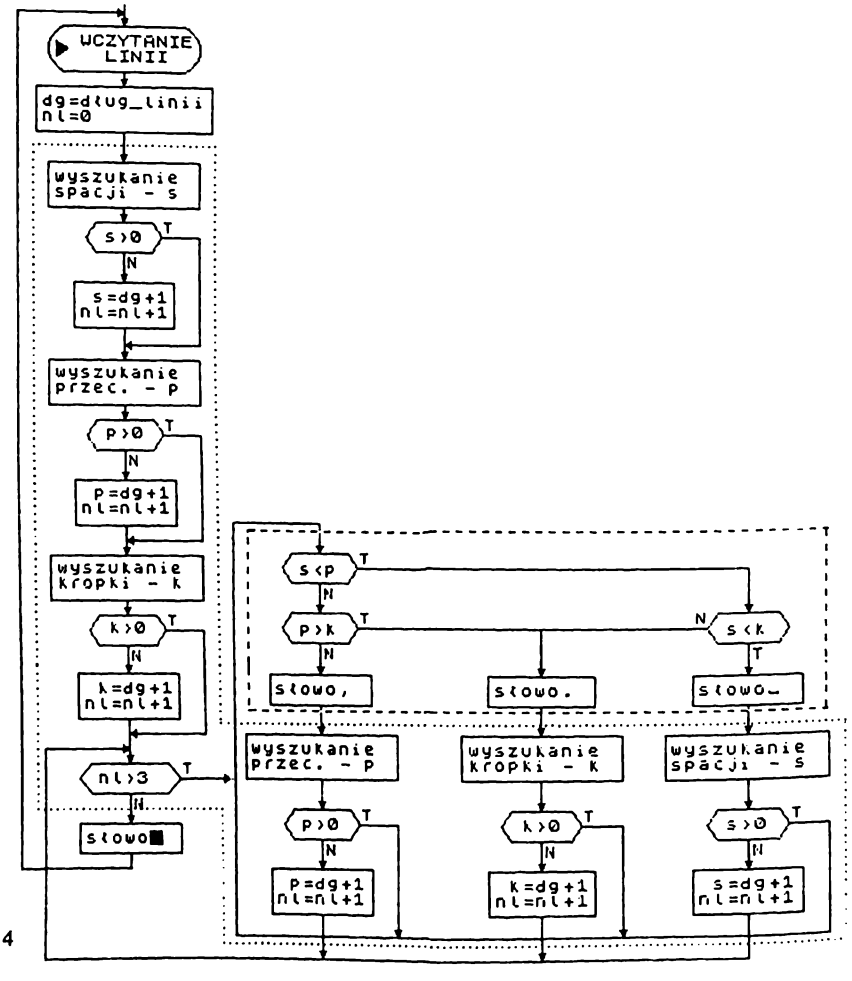
nie wykryto w tekście. Z kolei dla prawidłowego działania dalszej części algorytmu wystarczy wykryć tylko jaki jest pierwszy w kolejności separator. Stąd też nasuwa się rozwiązanie, w którym brak znaku w pozostałej części tekstu powinien być utożsamiany z wartością maksymalną pozycji w tekście, a więc na przykład równą LEN (I$)+1. Zapiszmy więc:

```
s=INSTR( i, l$, " " )
IF  s=0  THEN  s=LEN( l$ ) + 1
p=INSTR( i, l$,"," )
IF  p=0  THEN  p=LEN( l$ ) + 1
k=INSTR( i, l$,"." )
IF  k=0  THEN  p=LEN( l$ ) + 1
```

Teraz wystarczy tylko wykorzystać fragment tablicy decyzyjnej (zaznaczony na rys. 8.12 grubszą ramką). Fragment ten jest przetworzony na sieć działań pokazaną na rys. 8.14 (fragment ujęty w ramki z kresek przerywanych). Nie ma co ukrywać, że jest to znacznie prostsze rozwiązanie.

Tak, ale musimy jeszcze przeanalizować kilka problemów tutaj występujących.

1. Podczas reorganizacji algorytmu zgubiliśmy wykrywanie słowa znajdującego się na końcu linii. W nowej wersji będzie to sytuacja, gdy wszystkie trzy wartości zmiennych *s*, *p*, *k* będą równe LEN (*I$*)+1. Dla uproszczenia badania tego warunku wprowadzimy licznik *nl* zliczający dla ilu znaków separatora (spacji, przecinka i kropki) zmieniliśmy wynik funkcji INSTR. Rozwiązanie to jest zapisane w sieci działań na rys. 8.14 (fragment ujęty w ramki z kropek).



Rys. 8.14

2. Zauważmy też, że jeżeli określamy długość słowa na podstawie danego separatora, to w kolejnym znaku wystarczy począwszy od nowego miejsca w tekście poszukiwać tylko tego separatora. Również i tutaj dokonujemy modyfikacji wyniku funkcji i odpowiednio ze zmiennej nl. Korzystamy też tutaj z właściwości funkcji INSTR, że jeżeli wartość pozycji początku poszukiwania tekstu jest większa od długości (tutaj $i > \text{LEN}(/\$)$) to wtedy wartość funkcji jest też równa zero.

Omówiony dotychczas fragment algorytmu jest pokazany w postaci sieci działań na rys. 8.14 (możliwość wystąpienia sytuacji błędnej pomijam).

Możemy już przystąpić do zapisania rozwiązania w języku BASIC, gdyż reszta obliczeń jest już prosta. Sprawdźmy jednak jeszcze, czy nie można ulepszyć naszego rozwiązania, uwzględniając różne możliwe przypadki.

1. W zdaniu słowa mogą być oddzielone dwoma lub więcej spacjami. Oczywiście wtedy nie należy doliczać do liczby słów, słowa o długości równej zero. Funkcja INSTR (i, /$, "␣") w takim przypadku zwróci wartość równą i we fragmencie programu *zapis_nowego_słowa_zakończonego_spacją* możemy uwzględnić ten fakt

```
REM  zapis_nowego_slowa_zakonczonego_spacja
IF  s>i  THEN  ls=ls+1          'jezeli  s=i, slowo puste
                                ' pomijamy
  sd=sd+s-i                     'dla s=i otrzymamy  sd=sd
   i=s+i                        'przesuwamy o jedna pozycje
                                ' w prawo
```

2. Podobnie w zdaniu mogą wystąpić następujące układy znaków:

a) ␣, ␣.

b) , ␣ .␣

c) ␣ , ␣ ␣ . ␣

d) , , . . . , , .

W pierwszym przypadku (a) otrzymamy odpowiednio wartości $p=i$ lub $k=i$. We fragmentach programu dotyczących *zapisania_nowego_słowa_zakończonego_przecinkiem* lub *kropką* należy wprowadzić podobną instrukcję jak w poprzednim punkcie.

Przypadek drugi (b) został już rozwiązany w poprzednim punkcie, gdyż ten fragment programu nie pamięta co było wcześniej – spacja czy przecinek lub kropka.

Przypadek trzeci (c) jest złożeniem dwóch pierwszych.

Ostatni przypadek (d) jest już w zasadzie również uwzględniony, z dokładnością do sytuacji wystąpienia obok siebie dwóch kropek. Nasze dotychczasowe rozwiązanie przyjmie istnienie nowego zdania – o długości równej zero. Pominiemy ten problem jako mało prawdopodobny. Przystępujemy więc do zapisu tekstu programu.

Tekst programu

```
10  REM  ------------------------------------------------------
15  '                        ANALIZA TEKSTU
20  '    ------------------------------------------------------
25  '    Autor: W.Iszkowski
30  '    Data : 85/07/20
35  '    ------------------------------------------------------
100 REM  Instrukcja dla uzytkownika:
105 '    ------------------------------------------------------
110 PRINT STRING$(30,"-")
115 PRINT "PROGRAM ANALIZY TEKSTU"
120 PRINT STRING$(30,"-")
125 PRINT
130 PRINT "Program analizuje wprowadzony z klawiatury tekst"
135 PRINT "wyznaczajac  srednia dlugosc  slow wystepujacych"
140 PRINT "w tekscie oraz srednia dlugosc zdan."
142 PRINT
143 INPUT "Czy chcesz przeczytac instrukcje (t/n)"; a$
144 PRINT
145 IF  ASC(a$)=ASC("n")  THEN  500
146 PRINT
150 PRINT "Instrukcja uzytkowania programu"
155 PRINT
160 PRINT "1. Przeznaczony do analizy tekst nalezy wprowadzac"
165 PRINT "   jako pojedyncze linie zawierajace  od 1  do 254"
170 PRINT "   znakow i zakonczone znakiem RETURN (ENTER)."
175 PRINT
180 PRINT "2. Separatorami slow  w  tekscie moze  byc spacja,"
185 PRINT "   przecinek lub kropka konczaca zdanie."
190 PRINT
195 PRINT "3. W kazdej wprowadzanej linii moze byc czesc zda-"
200 PRINT "   nia, cale zdanie  lub kilka zdan - kazde zakon-"
205 PRINT "   czone znakiem kropki."
210 PRINT
215 PRINT "4. Ostatnie wprowadzone w linii slowo nie moze byc"
220 PRINT "   kontynuowane  w linii nastepnej."
230 PRINT
235 PRINT "5. Koniec  tekstu  przeznaczonego  do analizy jest"
240 PRINT "   znak   @   umieszczony na poczatku nowej linii."
245 PRINT
250 PRINT
499 '    ------------------------------------------------------
500 REM  Inicjalizacja wartosci zmiennych
505 '    ------------------------------------------------------
510   ls=0                          'licznik slow
515   lz=0                          'licznik zdan
520   ll=0                          'licznik linii
525   sd=0                          'suma dlugosci slow
```

```
999 '     ------------------------------------------------------------
1000 REM  Czesc glowna programu
1005 '    ------------------------------------------------------------
1010 PRINT "Prosze wprowadzac tekst do analizy"
1020 LINE INPUT 1$
1030 dg=LEN(1$)                                   'dlugosc tekstu
1040 IF  dg=0  THEN 1020                          'tekst pusty
1050 IF  ASC(1$)=ASC("@")  THEN  2010             'wyniki !
1060  i=1
1070 nl=0
1080 s=INSTR(i,1$," ")                           'pierwsza spacja ?
1090 IF  s>0  THEN  1120
1100    s=dg+1                                    'brak spacji
1110   nl=nl+1
1120 p=INSTR(i,1$,",")                           'pierwszy przecinek ?
1130 IF  p>0  THEN  1160
1140    p=dg+1                                    'brak przecinka
1150   nl=nl+1
1160 k=INSTR(i,1$,".")                           'pierwsza kropka ?
1170 IF  k>0  THEN  1200
1180    k=dg+1                                    'brak kropki
1190   nl=nl+1
1199 '
1200 REM  analiza kolejnych elementow tekstu
1201 '
1210 IF  nl<3  THEN  1300                         'istnieje jeden z sep.
1220   REM  zapis ostatniego slowa w linii
1230   IF  dg>=i  THEN  ls=ls+1
1240   sd=sd+dg+1-i
1250   ll=ll+1
1260   GOTO  1020                                 'nastepna linia
1300 IF  s<p  THEN  1410
1310 IF  p>k  THEN  1520
1320   REM  zapis slowa zakonczonego przecinkiem
1330   IF  p>i  THEN  ls=ls+1
1340   sd=sd+p-i
1350    i=p+1
1360   p=INSTR(i,1$,",")                          'kolejny przecinek
1370   IF  p>0  THEN  1300
1380      p=dg+1                                  'brak przecinka
1390     nl=nl+1
1400     GOTO  1200
1410 REM FROM 1300
1420 IF  s>k  THEN  1520
1430   REM zapis slowa zakonczonego spacja
1440   IF  s>i  THEN  ls=ls+1
1450   sd=sd+s-i
1460    i=s+1
1470   s=INSTR(i,1$," ")                          'kolejna spacja
1480   IF  s>0  THEN  1300
1490      s=dg+1                                  'brak spacji
1500     nl=nl+1
1510     GOTO  1200
1520 REM zapis slowa zakonczonego kropka
1530 IF  k>i  THEN  ls=ls+1
1540 sd=sd+k-i
```

```
1550  i=k+1
1560 lz=lz+1
1570 k=INSTR(i,l$,".")                          'kolejna kropka
1580 IF  k>0  THEN  1300
1590    k=dg+1                                  'brak kropki
1600   nl=nl+1
1610   GOTO  1200
1699 '
1770 REM KONIEC analizy kolejnych elementow tekstu
1999 '
2000 REM  ! Wyprowadzanie wynikow !
2010 PRINT "WYNIKI ANALIZY TEKSTU"
2020 PRINT
2030 PRINT "Tekst jest zlozony z";ll;"linii."
2040 PRINT
2050 PRINT "Liczba zdan:";lz;
2060 IF  lz=0  THEN  2080
2070 PRINT SPACE$(10);"Srednia dlugosc:";sd/lz
2080 PRINT
2090 PRINT "Liczba slow:";ls;SPACE$(10);"Srednia dlugosc:";sd/ls
2100 PRINT
2110 PRINT "Dziekuje"
2120 INPUT "Czy nastepny tekst (t/n)";a$
2130 IF  ASC(a$)=ASC("t")  THEN  500
2140 PRINT "KONIEC"
2150 '   ------------------------------------------------------------
2160 END
```

Uruchamianie programu

Dalej przedstawię wyciąg z zapisu historii uruchamiania programu, którego ostateczny tekst jest pokazany powyżej.

1. Wprowadziłem fragment tekstu programu do linii 1260. Fragment ten powinien rozpoznawać słowa zakończone znakiem końca linii. A więc spróbowałem zadając tekst testujący w postaci:

   wacek
   malgorzata
   krzysztof
   #

   W wyniku działania programu otrzymałem komunikat o błędzie (w postaci Line does not exist in 1050). Po sprawdzeniu okazało się, że program przeanalizował te cztery linie i chciał przejść do modułu wyprowadzającego wyniki, którego jeszcze nie było. W trybie bezpośrednim sprawdziłem następnie wartości istotnych zmiennych:

   ?sd, ll
   24 3

   a więc program zliczył 24 znaki w 3 liniach (zgadza się!).

2. Następnie wprowadziłem kolejne fragmenty tekstu programu do linii 1400, a więc mogłem już przetestować wykrywanie znaków przecinka:

   war, car, dar
   #

   i znowu wróciłem do linii 1050 z komunikatem o błędzie co, jak już wiedziałem, jest prawidłowe. Po sprawdzeniu wartości zmiennych:

   ?sd, 11, ls
   9 1 3

   a więc 9 liter, 1 linia, 3 słowa – czyli na razie dobrze.

3. Znowu wprowadziłem kolejny fragment programu – do linii 1770 i zacząłem go testować podobnymi jak poprzednio testami. Oto jeden z nich:

   w
   a
   a
   #
   ?sd, 11, ls
   3 3 0

   a więc błędna jest liczba policzonych słów. Dodatkowe testy wykazały, że przypadek taki następuje tylko wtedy gdy końcem słowa jest koniec linii, a słowa są jednoznakowe. Analizie poddałem więc wszystkie linie tekstu programu zwiększające wartość zmiennej *ls*. Okazało się, że w linii 1230 był warunek $dg > i$ co dla słów jednoznakowych zawsze jest fałszem. Po poprawieniu warunku na $dg >= i$ i powtórnym sprawdzeniu tego samego testu otrzymałem już wyniki prawidłowe.

4. Wprowadziłem resztę programu i podczas kolejnego testowania programu – dla testu nie zawierającego znaku kropki – otrzymałem sygnalizację próby dzielenia przez zero w linii 2050 (w pierwotnej wersji programu linia 2050 zawierała też treść linii 2070). No tak, program był więc nieodporny na brak zdań w analizowanym przez niego tekście. Stąd więc pojawiła się linia 2060 i 2070.

5. Dla testu w postaci:

   w
   a.
   c
   d.
   #

   otrzymałem wyniki w postaci:

   Tekst jest zlozony z 4 linii.

   Liczba zdan: 2 *Srednia dlugosc: 2*
   Liczba slow: 4 *Srednia dlugosc: 1*

Dziekuje
Czy nastepny tekst (t/n)?

6. Po kilku kolejnych testach wykryłem, że w teście:

w.w.w.
#

program wykrył tylko jedno zdanie, chociaż poprzednio bywało już dobrze. Dokładniejsza analiza wykazała, że błędne wyliczenie liczby zdań występuje tylko wtedy gdy są one w tej samej linii. A więc dlaczego kolejna kropka nie jest wykrywana? Analizując tekst odpowiedniego fragmentu programu (linie 1520–1699) wykryłem, że w linii 1570 był zapis *k*=INSTR (*i*, *I*$, " "), a więc program poszukiwał po wykryciu znaku kropki kolejnej spacji, a nie kropki. Po poprawieniu tej linii – wynik testu był już poprawny.

7. Po kilku dodatkowych testach założyłem, że mój program jest już poprawny i działa zgodnie z założeniami.
8. Przy okazji zacząłem się denerwować, że każde kolejne uruchomienie programu od początku powoduje wyświetlenie całej instrukcji dla użytkownika (którą już przecież znałem na pamięć). Stąd też dołączyłem do programu linie 142–146 wpychając je między istniejące (całe szczęście, że wystarczyło na nie numerów linii).

Tyle wyciągu z historii.

Komentarz końcowy

Przedstawiłem tutaj dość wyczerpująco opis procesu projektowania algorytmu i pisania programu oraz jego uruchamiania. Muszę przyznać się, że oprócz tego opisu wyrzuciłem jeszcze kilka kartek z próbami poszukiwań najlepszego rozwiązania – przecież nie tak trudnego w gruncie rzeczy zagadnienia. Niech więc przykład ten będzie ilustracją stopnia skomplikowania procesu tworzenia programu.

Styl programowania

Wykorzystywanie funkcji tekstowych. W podstawowym przetwarzaniu tekstów należy dążyć do wykorzystywania istniejących standardowo funkcji tekstowych. Można oczywiście przekształcić tekst z ciągu znaków na ciąg liczb zapisanych w tablicy – oczywiście korzystając z funkcji ASC i MID$, a następnie je przetworzyć i złożyć ponownie w tekst. Jednak efektywniejsze jest skorzystanie z funkcji, gdyż te są zaimplementowane w języku wewnętrznym – a więc czas ich działania jest maksymalnie skrócony.

Warto też korzystać z funkcji STRING$, zamiast przechowywać tekst złożony z identycznych znaków. Podobną uwagę można odnieść do tekstów

będących wielokrotnie argumentami wyrażeń. Takie teksty należy jak najszybciej umieścić w osobnej zmiennej typu tekstowego. Porównajmy więc dwa fragmenty programu wyprowadzającego opis pewnych wyników.

```
500  PRINT "------------------------------"
510  PRINT "  X  I  X  I  X  I  X  I  X  I"
520  PRINT "------------------------------"
530 '
540 '
550 '
600 PRINT  STRING$(30,"-")
610 LET    t$="  X  I"
620 PRINT  t$;t$;t$;t$;t$
630 PRINT  STRING$(30,"-")
```

Out of string space

Wyczerpanie obszaru przeznaczonego na teksty spowodowane zbyt dużą ich liczbą. Sytuacja ta świadczy o przepełnieniu pamięci. W starszych implementacjach systemu BASIC obszar przeznaczony na teksty był wydzielany z obszaru pamięci i musiał być deklarowany instrukcją.

CLEAR *wielkość_obszaru_na_teksty*

W tych przypadkach pojawienie się informacji o braku pamięci świadczy o zarezerwowaniu zbyt małego obszaru na teksty.

String formula too complex

Wyrażenie tekstowe jest za skomplikowane dla systemu BASIC. Wystąpienie tego komunikatu i przerwanie wykonywania programu świadczy o mniejszych możliwościach systemu BASIC, który się zagubił przy interpretacji wyrażenia. Teoretycznie, jeżeli wyrażenie tekstowe jest zapisane zgodnie z regułami składni to błąd ten nie powinien wystąpić. Poprawienie programu – interpretatora systemu BASIC nie możemy sami poprawić – polega na uproszczeniu zapisu wyrażenia przez podzielenie go na kilka prostszych wyrażeń. Przeanalizujmy zapis wyrażenia.

```
10  LET w$="WYDAWNICTWO NAUKOWO-TECHNICZNE"
20  LET w$=LEFT$(w$,12) + LEFT$(RIGHT$(w$,10),9) +
MID$(w$,19,2)+MID$(w$,13,6)+RIGHT$(w$,1)
30 PRINT w$
RUN
WYDAWNICTWO TECHNICZNO-NAUKOWE
Ok
```

Miałem nadzieję, że próba wykonania tego programu spowoduje wyprowadzenie komunikatu "String formula too complex", aby można było pokazać sposób uproszczenia tego wyrażenia. Niestety mój zaprzyjaźniony komputer zinterpretował wyrażenie poprawnie. Proszę jednak pamiętać, że inny system może być mniej sprawny.

Problemy

8.1. Podaj długość tekstu

```
a) ""abc"                    e)  SPACE$(10)
b) "      "                  f) RIGHT$( "TEKST",4 )
c) "121"                     g) CHR$(121)
d) "STR$(121)                h) ""
```

8.2. Podaj wartość wyrażenia relacji

```
a) t$ < LEFT$(t$,5)          e)  "0" < "ZERO"
b) "a" <= "A"                f)  "=" = CHR$(61)
c) "121" <> STR$(121)        g)  "$" < "z1"
d) SPACE$(5) < SPACE$(10)    h)  "-10" > "+10"
```

8.3. Podaj, które z instrukcji są błędne i dlaczego

```
a) LET  a$= "alfa"<"beta
b) LET  b$=LEFT$( "BASIC",4) + RIGHT$( "BASIC",4)
c) IF   a$+b$ > LEN(c$)  THEN  200
d) IF   MID$( "alfa",3 ) = LEFT$( RIGHT$( "alfa",2), 1 )  THEN  100
e) PRINT  alfa,beta
f) PRINT  al,fa,be,ta
```

8.4. Napisać program, który po wczytaniu trzech wartości, dwóch tekstowych i jednej liczbowej (w przykładach są podkreślone), wyprowadza następujące, prawidłowe gramatycznie zdania:

a) Student Jan Kowalski otrzymał 3 punkty.
   Studentka Anna Kowalska otrzymała 5 punktów.
   (Płeć rozpoznajemy po ostatniej literze imienia – jest tylko jeden wyjątek – Maria Kowalski).

b) Jedna kostka masła kosztuje 101 złotych.
   Dwie kostki masła kosztują 202 złote.
   ...

   (aż do dziesięciu)

c) To jest czarny kot.
   Tutaj nie ma czarnego kota.
   Przypatrz się temu czarnemu kotu.
   Nie baw się czarnym kotem (itd. według odmiany przez przypadki).

8.5. Napisać program, który dla sekwencji wprowadzanych (osobno) słów tworzy

a) skrót – np. Polskie Koleje Państwowe = PKP,
b) skróty złożone z dwóch, trzech, itd. liter – np. Widzewska Fabryka Maszyn = WiFaMa,
c) ich odpowiedniki wspak w ramach całej sekwencji słów oraz w ramach poszczególnych słów.

Let's learn BASIC English

char znak tekstu, napisu
string tekst, napis
concatenation konkatenacja, złożenie
string expression wyrażenie tekstowe
ASCII American Standard Code for Information. Interchange Amerykański Kod Standardowy do Przesylania Informacji
ISO International Organization for Standarization Międzynarodowa Organizacja Normalizacyjna
LINE INPUT WPROWADŹ LINIĘ
ASC KOD znaku (od ASCII)
LENgth DŁUGOŚĆ tekstu
VALue WARTOŚĆ
INSTR (IN STRing) W TEKŚCIE
**CHR$** ZNAK$
**LEFT$** LEWY$
**RIGHT$** PRAWY$
MID$ (MIDdle) ŚRODKOWY$
**SPACE$** SPACJA$
**STRING$** TEKST$
STR$ (STRing) TEK$

END of lesson

9. Funkcje i podprogramy

Zestaw funkcji standardowych w języku BASIC zawiera najbardziej podstawowe funkcje typu liczbowego i tekstowego. Jednak w wielu algorytmach korzysta się z innych funkcji, które stosuje się analogicznie jak funkcje rodzaju SIN czy LEFT$. W tym celu w BASICu umożliwiono definiowanie funkcji przez programującego. Ograniczeniem jest fakt, że wyrażenie obliczenia wartości funkcji musi być zawarte w jednej linii.

Analizując dokładniej przedstawione dotychczas programy, możemy zauważyć, że niektóre fragmenty tekstu są do siebie podobne, pod względem postaci i kolejności zapisu instrukcji. Różnice ograniczają się przy tym do innych nazw niektórych zmiennych oraz oczywiście do innej numeracji linii. Pamiętamy też o jednym z podstawowych postulatów dobrego stylu programowania – modularność struktury.

W języku BASIC możliwe jest wyodrębnienie fragmentów tekstu programu jako podprogramów, które mogą być wykonywane w różnych, wskazanych jawnie częściach sekwencji instrukcji programu. Konstrukcja ta uwzględnia oba wnioski z poczynionych wyżej spostrzeżeń – zmniejsza długość zapisu tekstu programu, tworząc jednocześnie przejrzystą strukturę modularną.

9.1. Definiowanie i wywoływanie funkcji

Funkcje standardowe są zdefiniowane bezpośrednio w języku BASIC. Pozostałe funkcje muszą zostać zdefiniowane w programie przed ich użyciem. Definicja funkcji polega na podaniu wzorca obliczenia wartości funkcji. Do definiowania funkcji służy *instrukcja definicji.*

DEF FN*identyfikator* (*parametry*) = *wyrażenie_liczbowe*
DEF FN*identyfikator*$ (*parametry*) = *wyrażenie_tekstowe*

Wykonanie instrukcji definicji polega na udostępnieniu systemowi BASIC wzorca obliczenia wartości funkcji. *Identyfikator*, konstruowany analogicznie jak dla zmiennych, rozszerza standardowy początek nazwy funkcji FN rozróżniając je. Uzupełnienie *identyfikatora* symbolem $ wyróżnia funkcje o wartościach typu tekstowego. Podane w nawiasach *parametry* formalne – zero, jeden lub więcej oddzielone przecinkami – wskazują, które ze zmiennych należą do dziedziny funkcji (są jej argumentami). Parametry mogą być typu liczbowego lub tekstowego.

Zasadę definiowania funkcji wyjaśnimy na przykładzie definicji funkcji obliczania wartości równania kwadratowego

$$10\ \text{DEF FNf(x)} = (a*x + b)*x + c$$

W tym przypadku parametr formalny o identyfikatorze x przestaje być identyfikatorem zmiennej, a staje się jedynie elementem zaznaczającym miejsce wstawienia wartości podczas obliczania funkcji. Fakt ten możemy zaznaczyć następująco:

$$10\ \text{DEF FNf(}\blacksquare\text{)} = (a*\blacksquare + b)*\blacksquare + c$$

Pozostałe argumenty wyrażenia są traktowane identycznie jak dla instrukcji przypisania.

Przeanalizujmy jeszcze następujące definicje funkcji:

```
10 DEF FNk1( r )    = 3.1458 * r * r        'powierzchnia kola,
15 '
20 DEF FNw (x,y)    = SQR( x*x + y*y )      'dlugosc wektora,
25 '
30                                          'pierwszy i ostatni znak,
40 DEF FNd$( t$ )   = LEFT$( t$,1 ) + RIGHT$( t$,1 )
45 '
50                                          'wstawienie po i-tym znaku
60 DEF FN(x$,y$,i) = LEFT$( x$,i ) + RIGHT$( y$,LEN( y$ )-1 )
```

Wszystkie te funkcje działają wyłącznie na parametrach formalnych. Jeżeli jednak na przykład linię 20 zapiszemy jako

$$20\ \text{DEF FNw1(x)} = \text{SQR}(x*x + y*y)$$

to wtedy tylko x będzie parametrem funkcji, a y będzie zmienną y o aktualnej wartości.

Instrukcja definicji funkcji musi być wykonana przed jej wykorzystaniem w programie. W większości realizacji systemu BASIC jest to jedyna instrukcja, która nie może być zleceniem wykonywanym natychmiast.

Wyliczenie wartości funkcji według zdefiniowanego wzorca następuje, za każdym razem, gdy jest argumentem obliczanego wyrażenia.

Wywoływanie funkcji definiowanej ma postać

FN*identyfikator* (*argumenty*)
FN*identyfikator*$ (*argumenty*)

Nazwą funkcji jest FN*identyfikator* lub FN*identyfikator*$. *Argumenty* w nawiasach są wyrażeniami o typach kolejno zgodnych z typami parametrów formalnych. Liczba argumentów, oddzielonych przecinkami, musi być równa liczbie parametrów formalnych.

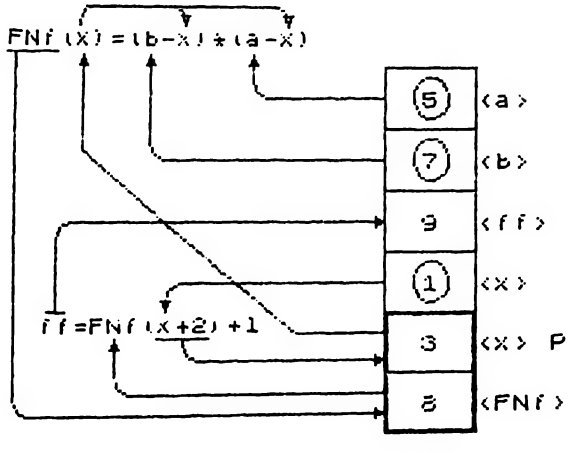
Rozważmy sposób wyliczenia wartości wyrażenia zawierającego wywołanie funkcji definiowanej.

```
10  DEF  FNf( x ) = (b-x) * (a-x)
15  '
60  LET  a=5
70  LET  b=7
80  LET  x=1
90  LET ff=FNf( x+2 ) + 1
100 PRINT ff
```

Na początek przyjmiemy, że w linii 100 zostaje wyznaczona wartość wyrażenia:

$(b-(x+2))*(a-(x+2)) =$
$(7-(1+2))*(5-(1+2)) =$
$(7-3)*(5-3) = 8$

Rozpatrzmy teraz schemat wykonania tych obliczeń przez system BASIC (rys. 9.1). Wartości zmiennych nadane w liniach 60, 70, 80, zostały zaznaczone kółkami. W wykonywaniu instrukcji w linii 90 zostanie najpierw obliczona wartość wyrażenia argumentu funkcji (tutaj 3), która staje się wartością parametru funkcji w obliczaniu wyrażenia definiującego funkcję. W przykładzie tym mamy więc dwa identyfikatory x, ale pełnią one zupełnie inne role.



Rys. 9.1

Kolejność obliczania wartości funkcji definiowanej

1. Obliczenie wartości wyrażeń kolejnych argumentów funkcji.
2. Przypisanie tych wartości kolejnym parametrom formalnym – ze sprawdzeniem zgodności typów.
3. Obliczenie wartości wyrażenia definiującego funkcję na podstawie bieżących wartości zmiennych i parametrów formalnych.
4. Przypisanie tej wartości nazwie funkcji będącej argumentem wyrażenia, w którym występuje.

Zachowując zasady wykorzystywania funkcji możemy też zapisać wywołanie typu

```
110  LET  fz=FNf( FNf( 3 ) + 1 )
```

które wylicza wartość wyrażenia w rozwiniętej postaci

$$PAR = ((b-3)*(a-3)) + 1$$
$$fz = (b-PAR)*(a-PAR)$$

czyli

$$PAR = 9 \qquad fz = 8$$

Ograniczenie zapisu wzorca wyliczania wartości funkcji do pojedynczej linii może czasem utrudnić zdefiniowanie nieco bardziej skomplikowanych obliczeń. Prostym rozwiązaniem jest zdefiniowanie funkcji pomocniczych. Dla wyrażenia znanego w elektrotechnice w postaci

$$I = \frac{E}{\sqrt{R^2 + \left(2\pi fL - \frac{1}{2\pi fC}\right)^2}}$$

zapis definicji funkcji może być następujący (*f* jest zmienną obliczeń):

```
30 DEF FNpi(ff,lc)    = 2 * 3.14158 * ff * lc
35 DEF FNwa(ff,ll,cc)= FNpi(ff,ll) - 1/FNpi(ff,cc)

40 DEF FNii( f, l, c ) = e / SQR( r*r + FNwa(f,l,c)*FNwa(f,l,c) )
```

Przy definiowaniu funkcji należy zawsze pamiętać, aby nie była to *definicja rekurencyjna*. Przykładem rekurencji nieskończonej jest znana piosenka-żart:

Było morze, w morzu kołek
A na kołku siedzi zając tak śpiewając
Było morze, w morzu...

Przekładając to na elementy programowania: nie wolno w wyrażeniu definiującym wartość funkcji umieszczać jej samej jako argumentu

```
50 DEF FNxxx(x) = x*x + FNxxx(x - 1)
```

Niedopuszczalna również jest rekurencja poprzez kilka wywołań funkcji. Zobrazujemy to historyjką

Jacek do Ali – pożycz gumkę, bo nie mam,
Ala do Krzysia – pożycz gumkę dla Jacka,
Krzyś do Maćka – pożycz gumkę dla Ali,
Maciek do Jacka – pożycz gumkę dla Krzysia,
Jacek (myśląc, że jest potrzebna druga gumka) do Ali – pożycz gumkę...

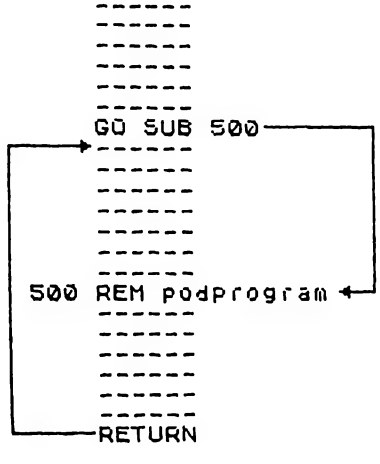
W naszym przypadku możliwość wystąpienia takiej sytuacji jest ograniczona składnią języka, gdyż przed wykorzystywaniem danej funkcji musi wcześniej być wykonana instrukcja jej definicji. Jest to warunek pierwszeństwa dynamicznego a nie statycznego.

Korzystając z funkcji standardowych możemy zdefiniować inne znane z matematyki funkcje – na przykład funkcje cyklometryczne korzystając z funkcji standardowej ATN(X) dla $X > 0$.

```
10  DEF  FNas( x )=ATN( x/SQR(1 - x*x))      'arcsin x
20  DEF  FNac( x )=ATN( SQR(1 - x*x)/x )     'arccos x
30  DEF  FNag( x )=3.14158 / 2 - ATN(x)      'arcctg x
```

9.2. Wykonywanie podprogramów

Podprogramem jest wyodrębniony logicznie fragment programu, którego wykonywanie może być zainicjowane z dowolnego punktu programu, do którego następuje powrót po zakończeniu wykonywania podprogramu. Ta nieco zawiła definicja podprogramu jest zilustrowana na rys. 9.2.



Rys. 9.2

Wywołanie podprogramu, czyli rozpoczęcie wykonywania instrukcji doń należących następuje poprzez *instrukcję przejścia do podprogramu.*

GOSUB *numer_linii*

Wykonanie instrukcji przejścia do podprogramu powoduje rozpoczęcie wykonywania instrukcji podprogramu, począwszy od podanego *numeru_linii.*

W odróżnieniu od instrukcji przejścia GOTO, zostaje zapamiętany ślad skąd nastąpiło to przejście. W podprogramie powrót do miejsca wywołania następuje *instrukcją powrotu.*

RETURN

Instrukcja powrotu z podprogramu powoduje przejście do wykonywania następnej, po instrukcji GOSUB, instrukcji wskazanej w zapamiętanym śladzie.

W podprogramie instrukcja powrotu może występować wielokrotnie, przy czym wykonana może być tylko jedna z nich. Przeanalizujmy fragment programu.

```
100  INPUT  x
110  GOSUB  1000
120  LET    x=7.1
130  GOSUB  1000
140  READ   x
150  GOSUB  1000
160  END
170  DATA   0
180  END
999  '
1000 REM  -----  podprogram  -----
1010 IF  x<=0  THEN  1040
1020   PRINT  "x dodatnie =";x
1030   RETURN
1040 IF  x=0   THEN  1070
1050   PRINT  "x ujemne   =";x
1060   RETURN
1070 PRINT "x rowne zero"
1080 RETURN
1090 END podprogramu
```

Po pierwsze w przykładzie tym warto zauważyć, że bez podprogramu tekst tego programu znacznie by się wydłużył. Z drugiej strony nie można tutaj zastosować instrukcji cyklu, jak to byłoby możliwe dla jednorodnego sposobu nadawania wartości zmiennej x.

W podprogramie występują trzy instrukcje powrotu. Zakładając wprowadzenie z klawiatury wartości 0, wykonanie programu będzie kolejnym wykonaniem instrukcji z linii:

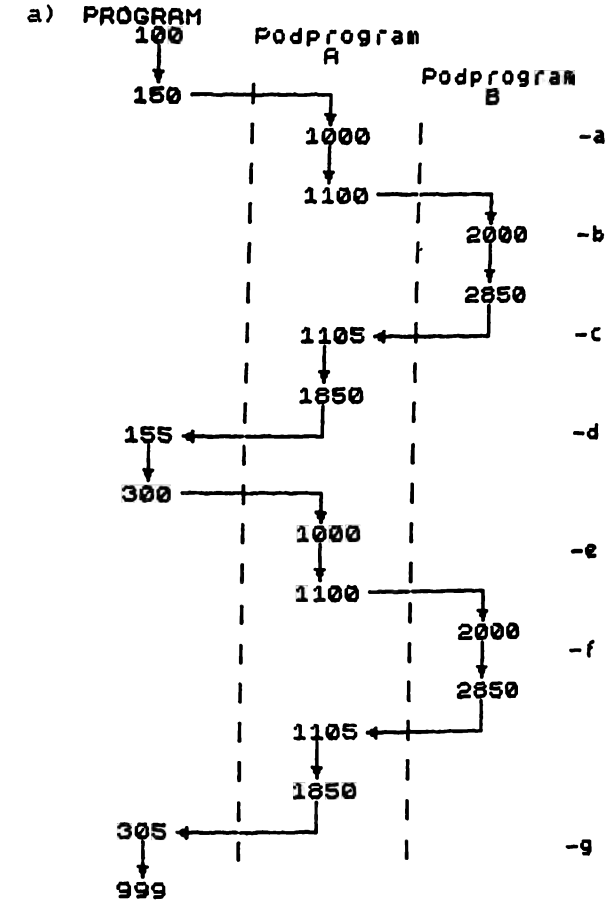
```
100   110   |   1000   1010   1040   1050   1060
120   130   |   1000   1010   1020   1030
140   150   |   1000   1010   1040   1070   1080
160         |
            |          instrukcje podprogramu
```

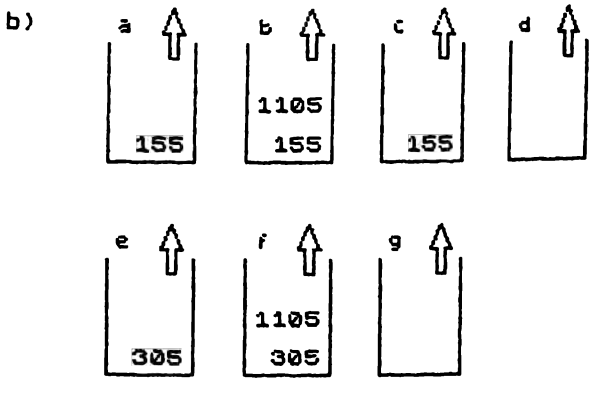
W trakcie wykonywania podprogramu jest też dozwolone wywołanie innego podprogramu. Struktura takiego programu może mieć postać

```
10   REM        * * Przyklad programu * *
100  REM ----------------------------------------
110  '            Czesc glowna programu
120  REM ----------------------------------------
130  '
140  '
150  GOSUB  1000                'wywolanie podprogramu  A
155  '...
160  '
300  GOSUB  1000                 'wywolanie podprogramu  A
305  '...
900  END  'czesc glowna podprogramu
999  '
1000 REM ----------------------------------------
1005 '                Podprogram  A
1010 REM ----------------------------------------
1015 '
1020 '
1100 GOSUB  2000                 'wywolanie podprogramu  B
1105 '...
1110 '
1850 RETURN
1900 END  'podprogram A
1999 '
2000 REM ----------------------------------------
2005 '                  Podprogram  B
2010 REM ----------------------------------------
2015 '
2020 '
2025 '
2850 RETURN
2900 END  'podprogram B
3000 END  'programu
```

W przykładzie wyróżnia się część główną programu oraz dwa podprogramy. Sekwencja wykonywanych instrukcji jest zilustrowana na rys. 9.3a. Ślad miejsca wywołania podprogramu jest zapamiętywany na *stosie*. Zawartość tego stosu dla kilku punktów wykonywanego programu jest pokazana na rys. 9.3b. Pojęcie stosu utożsamiamy tutaj ze stosem talerzy, z którego możemy zdejmować talerze tylko w kolejności odwrotnej do ich odkładania na stos (proszę sobie bowiem wyobrazić skutek wyjęcia talerza ze środka stosu).

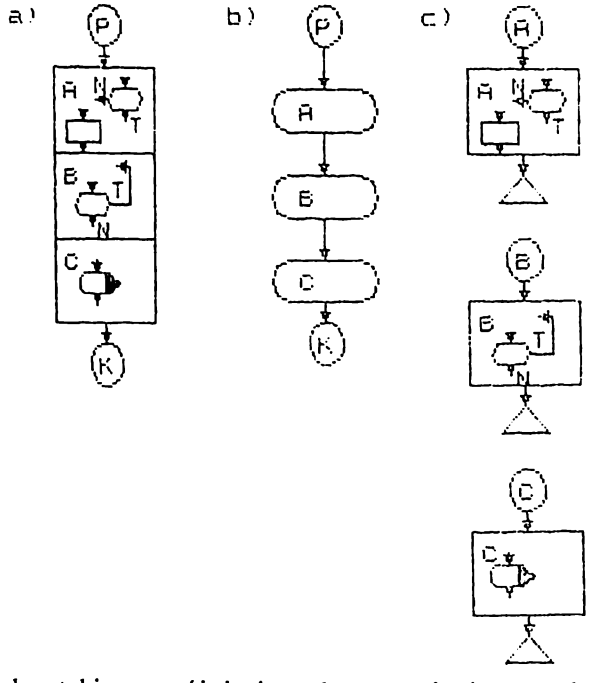
Podział tekstu programu na *część główną* oraz *podprogramy* jest dokonywany na podstawie logicznego wyróżnienia osobnych modułów realizowanego algorytmu. Podziału tego można już dokonać w trakcie projektowania algorytmu, szczególnie metodą od ogółu do szczegółu. Dokonane na pierwszym etapie wyróżnienie głównych części algorytmu pozwala realizować je następnie jako podprogramy.


a)
PROGRAM
100
Podprogram A
Podprogram B
150
1000
-a
1100
2000
-b
2850
1105
-c
1850
155
-d
300
1000
-e
1100
2000
-f
2850
1105
1850
305
-g
999


b)
a
155
b
1105
155
c
155
d
e
305
f
1105
305
g

Rys. 9.3

Dalej zajmę się zaprezentowaniem kilku typowych struktur programów z wyróżnionymi modułami czyli podprogramami. Dla programu o strukturze liniowej można rozdzielić poszczególne jego części kierując się ich logicznym przeznaczeniem oraz jednokrotnym powiązaniem sterowania (rys. 9.4). Typowym



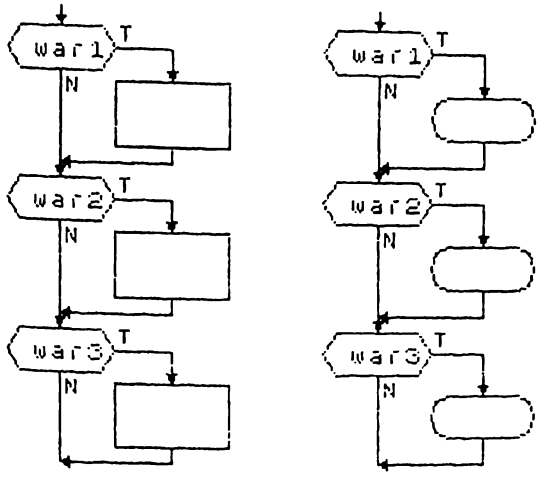
Rys. 9.4

przykładem takiego wyróżnienia podprogramów jest: rozdzielenie wprowadzania danych, przetwarzania (może w kilku etapach) oraz wyprowadzania wyników. W takim przypadku część główna programu będzie się składała z ciągu wywołań podprogramów.

```
10  REM  PROGRAM
20  GOSUB  100              'wprowadzanie danych
30  GOSUB  1000             'przetwarzanie - obliczenia
40  GOSUB  200              'wyprowadzanie wynikow
50  END
```

Kolejna możliwość, a czasem konieczność, wyróżnienia części programu jako podprogramu występuje w strukturze programu z rozgałęzieniami. Pamiętamy tutaj rozważania o prawidłowym i efektywnym zapisie instrukcji warunkowych. Często umieszczenie po fazie THEN tylko jednej instrukcji może być ominięte przez zastosowanie tam instrukcji przejścia do podprogramu. W typowym przypadku otrzymujemy zapis (rys. 9.5)

```
105  IF  war1  THEN  GOSUB  1100
110  IF  war2  THEN  GOSUB  1200
120  IF  war3  THEN  GOSUB  1300
```



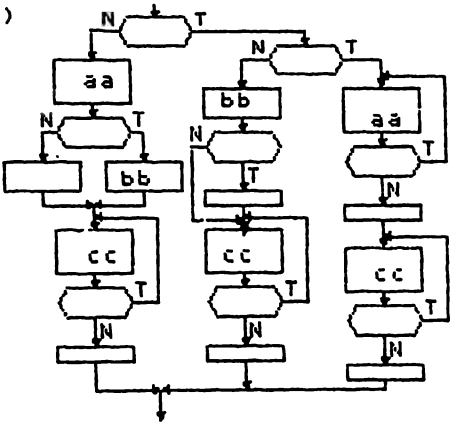
Rys. 9.5

Dotychczas omawiając strukturę programu z podprogramami traktowaliśmy je jako pojedyncze fragmenty treści programu, zapisane tylko w innych miejscach. Co więcej wprowadzając dodatkowe instrukcje przejścia do podprogramu i powrotu zwiększaliśmy długość tekstu programu – oczywiście zyskując na przejrzystości. Teraz pokażemy, że zastosowanie podprogramów może zmniejszyć długość tekstu programu. Dla rozbudowanych struktur programu z rozgałęzieniami często na różnych ścieżkach sterowania trzeba wykonać te same sekwencje instrukcji. Zastąpienie tych sekwencji podprogramami znacznie upraszcza zapis tekstu programu. W przekształcaniu tym zachowujemy dotychczasową strukturę sterowania (rys. 9.6). W niektórych przypadkach przy rozbudowanej strukturze rozgałęzionej możemy skorzystać z możliwości wielokrotnego rozmieszczenia w tekście podprogramu instrukcji powrotu RETURN (rys. 9.7).

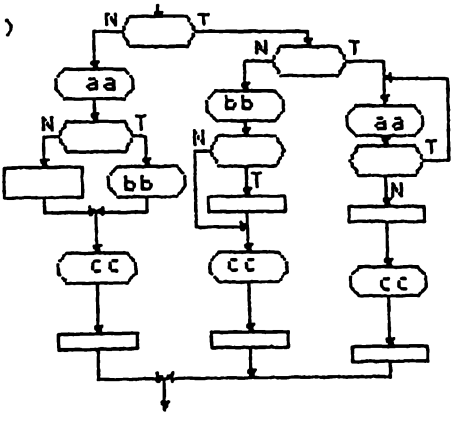
Wykorzystując pojęcie podprogramu możemy potraktować go jako pewien wzorzec obliczeń podobnie jak definicję funkcji. Problemem jest tutaj sparametryzowanie tych obliczeń, gdyż nie dysponujemy (na razie w omawianym tutaj języku BASIC) możliwościami, jakie dają parametry formalne dla funkcji. Jedynym rozwiązaniem jest przypisanie niektórym zmiennym, wykorzystywanym jako źródłowe (zawierające wartości), wartości przed wywołaniem podprogramu oraz pobranie z tych liczb lub innych zmiennych wartości obliczonych i dalsze ich wykorzystywanie zgodnie z logiką działania części głównej. Dokładniej problemy te przeanalizujemy w następnym podrozdziale.

Teraz wystarczy tylko stwierdzić, że analogicznie jak dla przypadków dotychczas omówionych możemy wykorzystywać podprogramy sparametryzowane. Z reguły konieczne jest wtedy poprzedzenie ich wywołania przypisaniem wartości parametrom oraz pobranie wartości wynikowych (rys. 9.8). Porównaj z pierwszym przykładem programu w tym rozdziale.

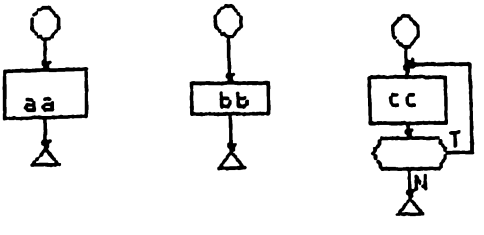
a)



b)

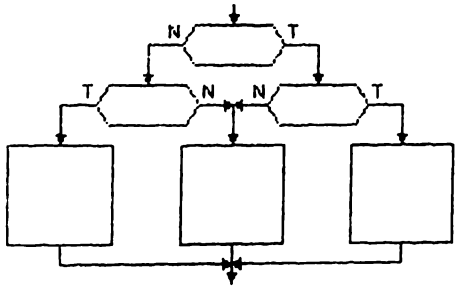


c)

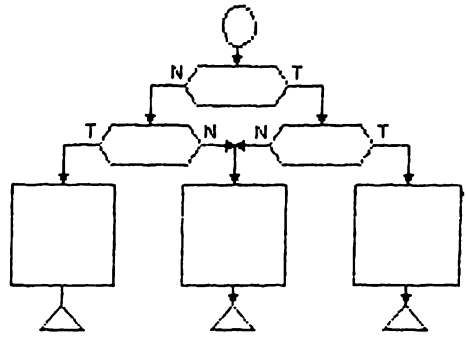


Rys. 9.6

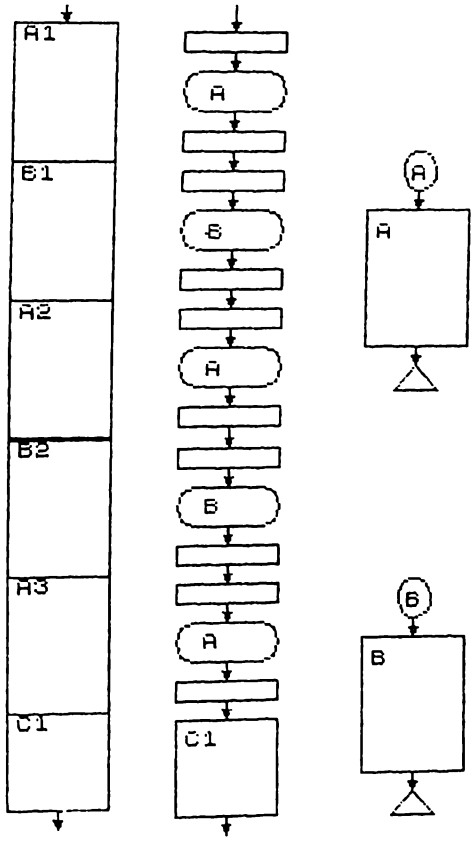
a)



b)



Rys. 9.7



Rys. 9.8

9.3. Współpraca z podprogramem

Instrukcje podprogramu i zawarte w nich zmienne są częścią całego programu. Wartości zmiennych ustalone przed wywołaniem podprogramu są dostępne w trakcie jego wykonywania. Po zakończeniu podprogramu dostępne są wszystkie zmienne z wartościami ostatnio przypisywanymi.

Podział logiczny zmiennych podprogramu. W zapisie instrukcji podprogramu wykorzystuje się:

1. *Parametry wejściowe*, będące zmiennymi, których wartości muszą być ustalone przed wywołaniem podprogramu.
2. *Parametry wyjściowe*, będące zmiennymi, którym wartości przypisano w trakcie wykonywania podprogramu, a które powinny być znane po jego zakończeniu.
3. *Zmienne lokalne*, które są wykorzystywane jedynie w obrębie wykonywanego podprogramu.
4. Niektóre *zmienne globalne* ogólnie dostępne w całym programie.

Wyjaśnienie zasad współpracy programu z podprogramem zaczniemy od najprostszej postaci.

```
10   INPUT  n
20   DIM  a(n)
30   '...
40   '...
100  GOSUB  1000
110  '...
999  END
1000 REM  ---  Wczytywanie wektora  ---
1010 FOR i9=0 TO n
1020   INPUT  a(i9)
1030 NEXT i9
1040 RETURN
```

W podprogramie wykorzystuję zmienne globalne: prostą n i tablicową $a(n)$ oraz zmienną lokalną *i9*. Podprogram w tej postaci można wykorzystywać tylko do wprowadzenia całej zawartości $(n+1)$-elementowego wektora a.

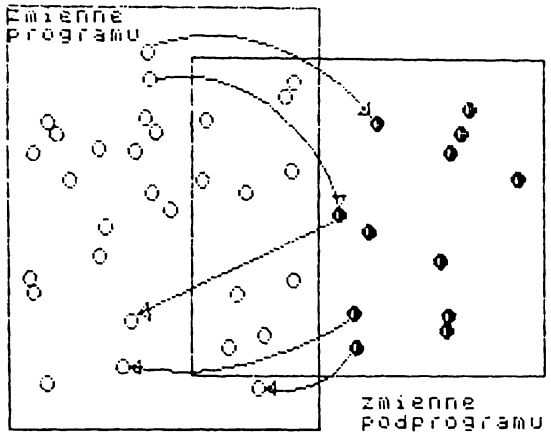
```
10  INPUT "Podaj rozmiar wektora";  n
20  DIM  a(n)
30  '
40  '
90  INPUT "Podaj zakres wprowadzania"; pd,pg
100 GOSUB  1000
110 '
120 '
990 END
999 '
1000 REM  - - - Wczytywanie fragmentu wektora - - -
1005 '                              sprawdzenie zakresow indeksow
1010  IF  pd<0  THEN  1050
1020  IF  pg>n  THEN  1050
1030  FOR i9=pd TO pg
1040    INPUT  a(i9)
1050  NEXT i9
1060  RETURN
1070  PRINT "indeksy poza zakresem ";pd;"  ";pg
1080  RETURN
```

W zmodyfikowanej wersji podprogramu możemy już wczytywać jego fragment określony wartościami parametrów wejściowych *pd* i *pg*. Pozostałe zmienne są wykorzystywane jak poprzednio – czyli jest to nadal podprogram dotyczący wektora *a*(*n*). Dokonamy jeszcze dalszej modyfikacji podprogramu

```
10  INPUT "Podaj rozmiar wektora";  n
20  DIM  a(n)
30  '
40  '
90  INPUT "Podaj zakres wprowadzania"; pd,pg
100 GOSUB  1000
110 IF  ps=0  THEN  PRINT "indeksy poza zakresem"
120 '
990 END
999 '
1000 REM  - - - Wczytywanie fragmentu wektora - - -
1010  LET  ps=0
1020 '                              sprawdzenie zakresow indeksow
1030  IF  pd<0  THEN  RETURN
1040  IF  pg>n  THEN  RETURN
1050  FOR i9=pd TO pg
1060    INPUT  a9
1070    LET     ps=ps + a9
1080    LET  a(i9)=a9
1090  NEXT i9
1100  RETURN
```

Wprowadziłem tutaj parametr wyjściowy *ps*, który pełni dwie funkcje. Dla błędnych wartości zakresu wprowadzania przyjmuje wartość równą 0, dla prawidłowych – sumę wartości elementów, co pozwala po zakończeniu podprogramu rozstrzygnąć, jakie były warunki jego wykonywania.

W podsumowaniu tych przykładów – współpraca programu z podprogramem polega na przygotowaniu dla niego wartości parametrów wejściowych, wywołaniu podprogramu, a następnie na dalszym przetwarzaniu uzyskanych wartości parametrów wyjściowych. Niektóre lub wszystkie parametry wejściowe mogą być też wyjściowymi. W podprogramie mogą być też wykorzystywane zmienne globalne, łącznie z przypisywaniem im nowych wartości oraz zmienne



oznaczenia:

○ - zmienne globalne
◐ - zmienne lokalne

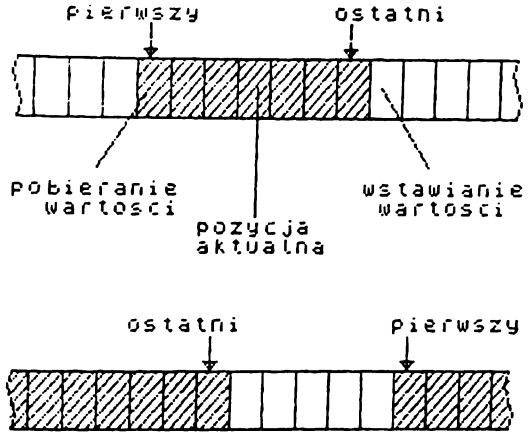
→◐ - parametr wejściowy
◐→ - parametr wyjściowy

Rys. 9.9

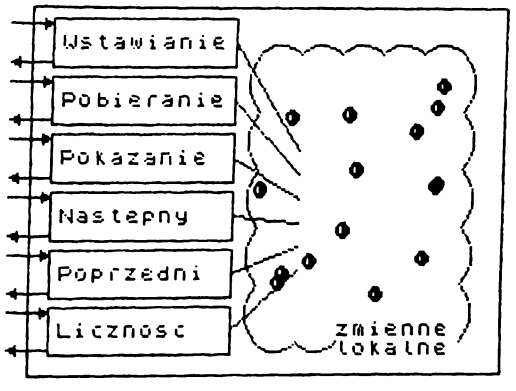
lokalne. Przyjmując dla przykładu istnienie tylko jednego podprogramu, rys. 9.9 przedstawia schematycznie podział logiczny (jedynie możliwy do stosowania w BASICu) wszystkich zmiennych programu. Podzbiory tych zmiennych nie są bowiem wzajemnie chronione. Nawet – tylko przypadkowo z części głównej programu lub z innego podprogramu mogą być zmodyfikowane wartości zmiennych lokalnych. Tym trudniejsze jest więc zadanie programującego, aby korzystając z różnych identyfikatorów zmiennych rzeczywiście je logicznie i fizycznie rozdzielić.

Przeanalizujmy jeszcze nieco bardziej rozbudowany pakiet podprogramów realizujących operacje na sekwencji danych. Jako definicję sekwencji danych przyjmiemy tutaj, że jest to uporządkowany ciąg wartości z określoną pozycją wartości pierwszej i ostatniej. Nowe wartości mogą być dołączone tylko na koniec tego ciągu oraz z ciągu mogą być pobierane kolejno wartości począwszy od pierwszej z ciągu. Istnieje też operacja pokazania wartości wskazanej jako aktualna, następnej oraz poprzedniej względem aktualnej. Ilustracją tego opisu jest rys. 9.10.

Do realizacji tego zbioru operacji wykorzystamy umownie zwany pakiet podprogramów. Cechą łączącą je jest to, że działają one na wspólnym zbiorze zmiennych lokalnych. Dodatkowo każdy z nich ma inne podzbiory parametrów wejściowych i wyjściowych. Ogólna struktura tego pakietu jest pokazana na rysunku 9.11.



Rys. 9.10



Rys. 9.11

```
999  STOP
1000 '-------------------------------------------------------
1005 '                 PAKIET PODPROGRAMOW
1010 '            OPERACJI NA SEKWENCJI DANYCH
1015 '-------------------------------------------------------
1020 '
1025 'Parametry  wejsciowe:
1030 '    n    - rozmiar wektora ( + 1 ) na sekwencje.
1035 '    pe   - wartosc wstawiana w sekwencje.
1040 'Parametry wyjsciowe:
1045 '    pw   - wartosc pobrana lub udostepniona z sekwencji.
1050 '    le   - liczba elementow w sekwencji.
1055 '.....................................................
1060 '
1100 REM                INICJACJA PAKIETU
1105 '
1110 DIM  qa(n)              'deklaracja wektora
1112 LET  qn=n               'rozmiar wektora
1115 LET  qn=n               'rozmiar wektora
1120 LET  qp=0               'pozycja pierwsza sekwencji
```

```
1125 LET  ql=-1              'pozycja ostatnia w sekwencji
1130 LET  qc=0               'liczba elementow w sekwencji
1135 LET  qa=0               'pozycja aktualna sekwencji
1140 RETURN
1145 END  'Inicjacji
1150 '.....................................................
1200 '
1205 REM   WSTAWIANIE ELEMENTU NA KONIEC SEKWENCJI
1210 '
1215 IF  qc)qn+1  THEN  1245
1220   LET   ql=ql + 1
1225   IF  ql)qn  THEN  LET .ql=0
1230   LET  qa(ql)=pe
1235   LET       qc=qc + 1
1240   RETURN
1245 PRINT "Brak miejsca w sekwencji"
1250 RETURN
1255 END  'Wstawiania
1260 '
1265 '.....................................................
1300 '
1305 REM   POBIERANIE ELEMENTU Z POCZATKU SEKWENCJI
1310 '
1315 IF  qc=0  THEN  1900
1320   LET  pw=qa(qp)
1325   LET  qc=qc - 1
1330   IF  qa=qp  THEN  LET qa=qa +1
1335   IF  qa)qn  THEN  LET qa=0
1340   LET  qp=qp + 1
1345   IF  qp)qn  THEN  LET qp=0
1350   RETURN
1355 END   'Pobierania
1360 '.....................................................
1400 '
1405 REM  POKAZANIE ELEMENTU AKTUALNEGO
1410 '
1415 IF  qc=0  THEN  1900
1420   LET  pw=qa(qa)
1425   RETURN
1430 END  'Pokazywania
1435 '.....................................................
1500 '
1505 REM  POKAZANIE ELEMENTU NASTEPNEGO
1510 '
1515 IF  qc=0  THEN  1900
1520   LET  qa=qa+1
1525   IF  qa)qn  THEN  LET qa=0
1530   LET  pw=qa(qa)
1535   RETURN
1540 END  'Pokazania nastepnego
1545 '.....................................................
1600 '
1605 REM  POKAZANIE ELEMENTU POPRZEDNIEGO
1610 '
1615 IF  qc=0  THEN  1900
1620   LET  qa=qa - 1
1625   IF  qa(0  THEN  LET qa=n
1630   LET  pw=qa(qa)
1635   RETURN
1640 END  'Pokazania poprzedniego
1645 '.....................................................
1700 '
1705 REM  POKAZANIE ELEMENTU PIERWSZEGO
1710 '
1715 IF  qc=0  THEN  1900
1720   LET  qa=qp
1725   LET  pw=qa(qa)
1730   RETURN
1735 END   'Pokazania pierwszego
1740 '.....................................................
```

```
1800 '
1805 REM  LICZNOSC ELEMENTOW W SEKWENCJI
1810 '
1815 LET  ll=qc
1820 RETURN
1825 END  'Licznosci
1830 '.....................................................
1900 '
1905 REM  KOMUNIKAT
1910 '
1915 PRINT "Brak elementu w sekwencji"
1920 RETURN
1925 END  'Komunikatu
1930 '.....................................................
1935 '-----------------------------------------------------
```

Przykład 9.1. „Analiza tekstu"

Opis problemu

Rozwiązanie podane w przykładzie 8.1 należy przekształcić do struktury programu modularnego z podprogramami.

Metoda rozwiązania

Nie będziemy się zajmować poszukiwaniem najlepszej struktury algorytmu. Naszym zadaniem jest podzielenie programu na moduły. Tutaj mamy ułatwione zadanie, gdyż dysponujemy już tekstem gotowego programu – napisanego zgodnie z dotychczas poznanymi zasadami poprawnego programowania. Poszukajmy więc w tekście fragmentów podobnych wskazując na ich różnice.
Pierwsza grupa instrukcji podobnych jest zawarta w liniach

1070–1100	"␣", *s*, 1110
1110–1150	" , ", *p*, 1160
1160–1180	" . ", *k*, 1200
1360–1390	" , ", *p*, 1300
1470–1500	"␣", *s*, 1300
1570–1600	" . ", *k*, 1300

Uwzględniając różnice, zapis tego fragmentu jest następujący:

```
3100  p2=INSTR(i, 1$, p1$)
3110  IF  p2>0  THEN  3140
3120    p2=dq+1
3130    n1=n1+1
3140  REM ENDIF
```

Zauważmy jeszcze, że w częściach programu z linii 1360–1390, 1470–1500 i 1570–1600 numer linii w instrukcji warunkowej jest identyczny oraz powoduje przejście poza ten fragment. Dokładniejsza analiza tekstu i algorytmu pozwala stwierdzić, że jest to skrócone, bezpośrednie przejście do linii 1300, zamiast

instrukcją GOTO 1200, 1210 i instrukcją IF $nl < 3$ THEN 1300. Możemy więc już zapisać ten fragment w postaci podprogramu określając parametry wejściowe, wyjściowe oraz zmienne lokalne (patrz tekst programu).

W podprogramie, dla zmniejszenia liczby bezpośrednio wymienianych parametrów i pewnego uproszczenia zapisu, wprowadziłem funkcję poszukiwania znaku o definicji

600 DEF FNzn(x$) = INSTR (i, l$, x$)

Przed wywołaniem podprogramu należy ustalić wartość parametru *p1$* = "␣" lub "." lub ",". Po zakończeniu podprogramu wartość parametru *p2* należy przypisać odpowiednio zmiennej *s* lub *k* lub *p*.

Druga grupa instrukcji podobnych jest zawarta w liniach

1320–1400	",", *p*
1430–1510	"␣", *s*
1520–1610	".", *k*

Uwzględniając różnice, zapis tego fragmentu jest następujący:

```
4090 REM  zapis-slowa-zakonczonego-????
4100 IF  p2>i  THEN  ls=ls+1
4110   sd=sd+p2-i
4120    i=p2+1
4130   GOSUB  3000
```

Dla podprogramu wywołanego w linii 4130 parametr wejściowy jest równocześnie parametrem dla tego nowego podprogramu. Również jego parametr wyjściowy jest tym samym parametrem.

Tekst programu

```
10  REM  ------------------------------------------------
15  '                    ANALIZA TEKSTU
17  '            z wykorzystaniem podprogramow
20  '    ------------------------------------------------
25  '    Autor: M.Iszkowski
30  '    Data : 85/07/20
33  '    Uwaga: Przerobiona wersja programu z przykladu  8.1
35  '    ------------------------------------------------
100 REM  Instrukcja dla uzytkownika:
105 '    ------------------------------------------------------
110 '
115 '    Instrukcje dla uzytkownika dla skrocenia tekstu usunieto
120 '
499 '     ------------------------------------------------------
500 REM  Inicjalizacja wartosci zalenych
505 '    ------------------------------------------------------
510   ls=0                          'licznik slow
515   lz=0                          'licznik zdan
520   ll=0                          'licznik linii
525   sd=0                          'suma dlugosci slow
595 '                               Definicja funkcji
600   DEF FNzn( z$ ) = INSTR(i,l$,z$)
605 '
```

```
999 '    ------------------------------------------------------------
1000 REM  Czesc glowna programu
1005 '    ------------------------------------------------------------
1010 PRINT "Prosze wprowadzac tekst do analizy"
1020 LINE INPUT l$
1030 dg=LEN(l$)                                  'dlugosc tekstu
1040 IF  dg=0  THEN 1020                         'tekst pusty
1050 IF  ASC(l$)=ASC("0")  THEN  2010            'wyniki !
1060  i=1
1070 nl=0
1080  p1$=" "                                    'pierwsza spacja ?
1090  GOSUB  3000
1100  s=p2
1120  p1$=","                                    'pierwszy przecinek ?
1130  GOSUB  3000
1140  p=p2
1160  p1$="."                                    'pierwsza kropka ?
1170  GOSUB  3000
1180  k=p2
1199 '
1200 REM  analiza kolejnych elementow tekstu
1201 '
1210 IF  nl<3  THEN  1300                        'istnieje jeden z sep.
1220   REM  zapis ostatniego slowa w linii
1230   IF  dg>=i  THEN  ls=ls+1
1240   sd=sd+dg+1-i
1250   ll=ll+1
1260   GOTO  1020                                'nastepna linia
1300 IF  s<p  THEN  1410
1310 IF  p>k  THEN  1520
1320   REM  zapis slowa zakonczonego przecinkiem
1330   p1$=","
1340   p2=p
1350   GOSUB  4000
1360   p=p2
1400   GOTO  1200
1410 REM FROM 1300
1420 IF  s>k  THEN  1520
1430   REM zapis slowa zakonczonego spacja
1440   p1$=" "
1450   p2=s
1460   GOSUB 4000
1470   s=p2
1510   GOTO  1200
1520 REM zapis slowa zakonczonego kropka
1530  p1$="."
1540  p2=k
1550  GOSUB  4000
1560 lz=lz+1
1570  k=p2
1610   GOTO  1200
1699 '
1770 REM KONIEC analizy kolejnych elementow tekstu
1999 '
2000 REM  ! Wyprowadzanie wynikow !
2010 PRINT "WYNIKI ANALIZY TEKSTU"
2020 PRINT
2030 PRINT "Tekst jest zlozony z";ll;"linii."
2040 PRINT
2050 PRINT "Liczba zdan:";lz
2060 IF  lz=0  THEN  2080
2070 PRINT SPACE$(10);"Srednia dlugosc:";sd/lz
2080 PRINT
2090 PRINT "Liczba slow:";ls;SPACE$(10);"Srednia dlugosc:";sd/ls
2100 PRINT
2110 PRINT "Dziekuje"
2120 INPUT "Czy nastepny tekst (t/n)"; a$
2130 IF  ASC(a$)=ASC("t")  THEN  500
2140 PRINT "KONIEC"
2150 '    ------------------------------------------------------------
2160 END
```

```
2999 '
3000 '  ------------------------------------------------------------
3005 REM SUBPROGRAM    Poszukiwanie kolejnego separatora
3010 '  ------------------------------------------------------------
3015 '
3020 ' Parametr wejsciowy  :  pl$  -separator
3025 ' Zmienne globalne    :  dg    -dlugosc tekstu
3030 '                        nl    -liczba brakujacych separatorow
3035 ' Parametr wyjsciowy  :  p2    -pozycja separatora
3040 ' Zmienne lokalne     :         (brak)
3045 '
3100  p2 = FNzn( pl$ )
3110  IF  p2>0  THEN  RETURN
3120    p2=dg+1
3130    nl=nl+1
3140  REM ENDIF
3150  RETURN
3200  END             'Poszukiwanie kolejnego separatora
3205 '
4000 '  ------------------------------------------------------------
4005 REM SUBPROGRAM     Zapis slowa zakonczonego separatorem
4010 '  ------------------------------------------------------------
4015 '
4020 ' Parametr wejsciowy  :  p2    -pozycja separatora
4025 ' Zmienne globalne    :  ls    -liczba slow
4030 '                        sd    -suma dlugosci slow
4035 '                        pl$   -
4040 '                        dg    -  (dla podpr. 3000)
4045 '                        nl    -
4050 ' Parametr wyjsciowy  :  p2    -nowa pozycja separatora
4055 ' Zmienne lokalne     :  i     -poczatek poszukiwania
4060 '
4100 IF  p2>i  THEN  ls=ls+1
4110 sd=sd+p2-i
4120  i=p2+1
4130 GOSUB  3000
4140 RETURN
4200 END              'Zapis slowa zakonczonego separatorem
```

Styl programowania

Modularność programu jest podstawową cechą dobrego stylu programowania. Hasło to powinno przyświecać każdemu konstruującemu większy (złożony powiedzmy z ponad 50 linii) program.

Język BASIC w stosunku do innych języków programowania wysokiego poziomu ma dość ubogie mechanizmy umożliwiające strukturalizację programu.

Dlatego tym bardziej należy tutaj maksymalnie wykorzystać te możliwości stosując następujące podstawowe zasady:

1. Wyrażenia identyczne co do kolejności i liczby operacji, należy zastępować definicjami funkcji odpowiednio sparametryzowanymi. Nie należy też unikać definiowania funkcji dla wspólnych podwyrażeń. Z drugiej strony obliczanie funkcji dla tych samych wartości parametrów należy sprowadzić do pojedynczego odwołania.

Zamiast więc zapisu w postaci (algorytm tych obliczeń jest wydumany)

```
100  LET  x = p*p - 3.14158*z/y + 1/(3.14158*p/r)
110  LET  y = r*r - 3.14158*x   + 1/(3.14158*x)
120  LET  z = EXP(y) * EXP(y) - x + 1/x
```

lepiej jest zapisać ten sam fragment w postaci

```
1000  DEF FNa1(p1,p2) = 3.14158 * p1 / p2
1005  '
1010  DEF FNws(s1,s2,s3) = s1*s1 - s2 + 1/s3
1015  '
1100  LET  x = FNws( p, FNa1(z,y), FNa1(p,r) )
1109  LET px = FNa1( x, 1 )
1110  LET  y = FNws( r, px, px )
1120  LET  z = FNws( EXP(y), x, x )
```

Drugi zapis wydaje się z początku nieco bardziej skomplikowany, ale przy okazji przekształcania tekstu programu z definiowaniem funkcji zyskujemy:

a) lepsze zrozumienie naszego własnego zapisu obliczeń,
b) rzetelniejsze sprawdzenie poprawności zapisu obliczeń,
c) możliwość zoptymalizowania obliczeń wielokrotnych,
d) możliwość szybszego przetestowania działania funkcji dla kilku lub kilkunastu wyrażeń.

2. Programy o dużej liczbie linii należy dzielić na moduły w postaci podprogramów. Podział na podprogramy powinien następować w miejscach rozróżniających logicznie kolejne, dobrze określone czynności algorytmu. Wydzielone podprogramy można uruchamiać osobno testując je dla specjalnie dobranych danych, po czym dopiero umieszczać wspólnie w jednym programie. Nic nie stoi na przeszkodzie, aby część główna programu była złożona z sekwencji wywołań podprogramów

```
100  REM     PROGRAM    "xxxxxxxxx"
110  GOSUB   1000                    'opis dzialania programu
120  GOSUB   2000                    'inicjacja zmiennych
130  GOSUB   3000                    'wprowadzanie danych
140  GOSUB   4000                    'przetwarzanie danych
150  GOSUB   5000                    'wyprowadzanie wynikow
160  END
```

Taki zapis uzupełniony na przykład instrukcjami wielokrotnego wykonywania programu dla kolejnych zestawów danych jest bardzo przejrzysty i pozwala czytającemu poznać program od ogółu do szczegółu.

3. Podstawowym celem stosowania podprogramów jest możliwość wyodrębnienia z tekstu programu fragmentów podobnych algorytmicznie. Różnice w tych fragmentach stanowią podstawę do wyznaczenia zbioru parametrów wejściowych i zbioru parametrów wyjściowych. Sparametryzowane fragmenty tekstu programu jako podprogramy stają się osobnymi modułami.

Przy stosowaniu tego typu podprogramów zyskujemy:

a) zmniejszenie zajętości pamięci na treść programu, tym większe im częściej dany podprogram musi być wykorzystywany w różnych miejscach tekstu programu,

b) uproszczenie analizy działania programu poprzez rozróżnienie logiki sterowania globalnego od sterowania obliczeniami lokalnymi.

c) możliwość uporządkowania zbioru zmiennych i dokładniejszego przeanalizowania zakresu ich wykorzystywania – co ma istotny wpływ na łatwość wprowadzenia poprawek i modyfikacji programu.

Oczywiście stosowanie funkcji definiowanych oraz podprogramów z koniecznością wstępnego ustalania wartości parametrów wejściowych, a potem pobierania wartości parametrów wyjściowych, wprowadza dodatkowe opóźnienia zwiększając nieco czas działania programu. Również czas zużyty na logiczne podzielenie programu na podprogramy, wstępne określenie ich postaci oraz zestawu parametrów jest dłuższy od bezpośredniego napisania programu.

Trzeba jednak tutaj stwierdzić, że straty te są niewspółmiernie małe w stosunku do korzyści, jakie uzyskujemy przy uruchamianiu programu, wyszukaniu i poprawieniu w nim błędów, a także modyfikacji. Niestety większość programujących przekonuje się o tym dopiero na podstawie własnych doświadczeń, kiedy to po kilku nieudanych próbach poprawienia jednomodułowego bardzo długiego programu przystępowało do ponownego jego napisania – już wtedy zgodnie z zasadami dobrego stylu programowania.

Zapis tekstu podprogramu musi być czytelny i przejrzysty. Znowu rozpoczynamy od stwierdzenia, że język BASIC ma dość ubogie mechanizmy zapisu w pełni logicznie wyodrębnionego modułu. Stąd też szczególną uwagę trzeba zwrócić na następujące postulaty czytelnego zapisu treści podprogramu:

1. Fragmenty programu będące podprogramami powinny być łatwo widoczne przez przypisanie im wyróżnionych zakresów numeracji linii (np. 1000–1999, 2000–2999 itd.) oraz odkreślone instrukcją komentarza typu

```
1000  REM  ----------------------------------------------------
2000  '****************************************************
3000  '....................................................
4000  ':::::::::::::::::::::::::::::::::::::::::::::::::::
5000  REM  ****************************************************
```

(istnienie automatycznej repetycji znaku przy dłużej wciśniętym klawiszu znacznie upraszcza to zadanie).

2. Na początku podprogramu powinien znajdować się w komentarzach jego tytuł zawierający słowa precyzyjnie go identyfikujące. Podprogramy przeznaczone dla innych użytkowników powinny zawierać dodatkowo krótki komentarz opisujący ich funkcje i zasady działania.

3. Konieczne jest umieszczenie w komentarzach (przynajmniej skróconego) opisu zestawu parametrów wejściowych, wyjściowych oraz wykorzystywanych w podprogramie zmiennych globalnych i lokalnych. Identyfikatory parametrów oraz zmiennych lokalnych powinny być wyróżnione jednolitym znakiem. Przykładowo może to być zestaw typu

	p1	r1	l1	i1	s1	w1(50)	c1$
lub	qp	qr	ql	qi	qs	qw(30)	qc$

4. Wszystkie definicje funkcji powinny być zgrupowane na początku programu jako wyróżniony fragment, gdyż istnieje duże prawdopodobieństwo ich możliwych modyfikacji i wymiany.
5. Również podprogramy powinny być osobno zgrupowane na początku tekstu programu (wtedy należy je obejść instrukcją 1 GOTO *XXXXX*) lub na końcu (wtedy należy je poprzedzić instrukcją END).

Sytuacje błędne

Undefined user function

Brak definicji funkcji czyli funkcja została wywołana zanim została wykonana instrukcja zawierająca jej definicję. Prościej – systemowi BASIC nie jest znane jak daną funkcję obliczyć. Poprawienie tego błędu nie powinno z reguły nastręczać trudności. Czasem może się okazać, że próbowaliśmy zdefiniować funkcję rekurencyjnie.

Illegal function call

Niedozwolone, błędne wywołanie funkcji polegające na podaniu parametrów aktualnych niezgodnie co do kolejności i typu z parametrami formalnymi podanymi w definicji funkcji. No cóż – błąd ten należy szybko poprawić.

RETURN without GOSUB

Próba wykonania instrukcji powrotu, gdy brak zapamiętanego śladu. Ślad powrotu jest zapisywany instrukcją przejścia do podprogramu. Błąd ten występuje z reguły wtedy, kiedy zaczęliśmy wykonywać instrukcje należące do podprogramu bezpośrednio, a nie poprzez wykonanie instrukcji przejścia do podprogramu GOSUB. Przypadek taki czasem powstaje, gdy części głównej programu nie zakończyliśmy instrukcją końca END. Wtedy to po wykonaniu ostatniej instrukcji programu zaczyna się wykonywać podprogram (już raczej bez sensu). Przy okazji nie warto próbować stosować „uproszczeń" typu traktowania podprogramu raz jako fragmentu programu, a w innym przypadku jako podprogramu. Nie będę tutaj bliżej wyjaśniał tej konstrukcji. Ci co je zastosowali już i tak pewnie zrozumieli, jakie to niesie kłopoty w panowaniu nad logiką programu.

Problemy

9.1. Dla podanej definicji funkcji oraz wywołania wyznacz wartość wyniku.

```
10  DEF FNf(x,y) = a*x - b*y
20  LET  a=6
25  LET  b=5
30  LET  x=2
35  LET  y=3
```

```
a)  50  LET  z=FNf( 4,3 )
b)  50  LET  z=FNf( 0,0 )
c)  50  LET  z=FNf( 1,1 )
d)  50  LET  z=FNf( 1,-1 )
e)  50  LET  z=FNf( a,b )
f)  50  LET  z=FNf( b,a )
g)  50  LET  z=FNf( x,y )
h)  50  LET  z=FNf( x,x )

i)  50  LET  z=FNf( FNf( 3,2 ),FNf( 0,-1 ))
j)  50  LET  z=FNf( FNf( b,b ),FNf( y,y  ))
k)  50  LET  z=FNf( FNf( 2,FNf( 2,1 )),FNf( FNf( a,b ),y ))
```

9.2. Napisać definicje funkcji obliczających:
a) pole trójkąta przy znajomości długości trzech boków,
b) wartości funkcji hiperbolicznych,
c) literę alfabetu o danym numerze (A = 1, Z = 26),
d) wartość kodu znaku o danej wartości pozycji,
e) objętość stożka,
f) odległość między dwoma punktami o danych współrzędnych przestrzennych (x, y, z).

9.3. Podać wartość zmiennej z po wykonaniu programu zawierającego podprogram

```
100  REM  PRZYKLAD_A
110  LET  n=10
120  LET  i=2
130  GOSUB  500
140  PRINT  z
160  '
200  REM  PRZYKLAD_B
210  LET  n=10
220  LET  i=4
230  GOSUB  500
235  GOSUB  500
240  PRINT  z
260  '
300  REM  PRZYKLAD_C
310  LET  n=5
320  LET  i=0
330  GOSUB  500
340  LET  i=z
350  LET  n=20
360  GOSUB  500
370  PRINT  z
```

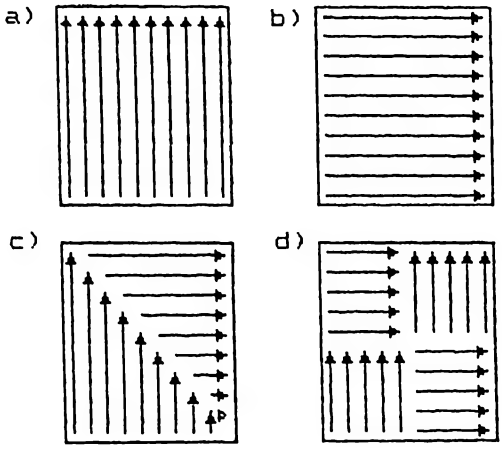
```
390  '
400  REM  PRZYKLAD_D
410  LET  z=10
420  LET  n=z
430  LET  i=-z
440  GOSUB  500
450  LET  k=z
460  GOSUB  500
470  PRINT  z
490  '
499  END
500  REM   ---- PODPROGRAM  ----
501  '
510  LET  z=0
520  FOR k=i TO n
530    LET  z=z+k
540  NEXT k
550  RETURN
560  '
```

9.4. Wyjaśnij, które z podanych fragmentów programów są błędne i dlaczego

```
a) 50  DEF FNz(z) = ( z$z + FNz(z) )

b) 50  DEF FNa(x) = FNb(x) - FNb(c)
   55  DEF FNb(y) = FNa(y) + FNa(c)

c) 50  DEF FNf(y) = y+1
   55  DEF FNg( FNf(y) ) = FNf(2) - FNf(1)

d) 50  DEF FNh(i,j,k) = ( a(i) + a(j) + a(k) )/3

e)   5  LET  x=10
    10  GOSUB  100
    20  PRINT  x
    30  GOSUB  10
    40  RETURN
   100  LET  x=2$x
   110  RETURN

f)   5  DIM  a(10)
    10  LET  i=1
    20  GOSUB  100
    30  LET  j=5
    40  GOSUB  100
   100  PRINT  a(i)
   110  RETURN

g)  10  LET  x=y+1
    20  RETURN
    30  LET  y=10
    40  GOSUB  10
    50  PRINT  x
    60  END

h)  10  LET  i=10
    20  LET  k=5
    30  GOTO 10
    40  PRINT  l
    50  END
   100  LET  l=i+k
   110  RETURN
```

9.5. Należy tak zaadaptować zapis algorytmu sortowania bąbelkowego, aby stanowił on podprogram realizujący sortowanie dowolnego wiersza macierzy w określonych granicach kolumn. Następnie należy napisać podobny podprogram do sortowania w kolumnach. Oba podprogramy należy wykorzystać do posortowania macierzy jak na rys. 9.12.

9.6. Należy przejrzeć wszystkie dotychczas napisane i uruchomione programy starając się je przerobić z maksymalnym wykorzystaniem definicji funkcji i podprogramów.



Rys. 9.12

Let's learn BASIC English

function funkcja
subroutine podprogram
parameter parametr
local variable zmienna lokalna
global variable zmienna globalna
sequence sekwencja
DEFinition DEFinicja
FN(FuNction) FuNkcja
GO SUBroutine IDŹ DO PODprogramu
RETURN WRÓĆ

END of lesson

10. Uruchamianie programu

Dotychczas nasza uwaga była skupiona głównie na tym, jak pogodzić ze sobą zaprojektowanie algorytmu dla danego problemu z możliwościami jego zapisu w języku BASIC. Naszym dążeniem było uzyskanie poprawnego merytorycznie i technicznie tekstu programu, uwzględniając w miarę możliwości dobry styl programowania. Niewiele jednak uwagi poświęciliśmy technice uruchomienia programu, a więc jego wprowadzenia do komputera, wykonania oraz przetestowania dla stwierdzenia jego poprawności i zgodności z założeniami algorytmu rozwiązywanego problemu.

Wspomnieliśmy już o tym, że prawdopodobieństwo uzyskania programu bezbłędnego natychmiast po jego wprowadzeniu do komputera jest praktycznie bliskie zeru. Stąd też, już na etapie konstruowania algorytmu oraz jego zapisu w języku programowania musimy się przygotowywać do metodycznego poszukiwania „istniejących" błędów.

Klasy błędów programu. W uruchamianym programie mogą wystąpić błędy należące do jednej z trzech klas:

1. *Błędy składniowe* polegające na niezgodnym ze składnią języka zapisem instrukcji (popularnie są to błędy ortograficzne).
2. *Błędy wykonywania* polegające na żądaniu wykonania niedopuszczalnej w danym kontekście operacji.
3. *Błędy logiczne* polegające na poprawnym funkcjonowaniu programu, ale wykonującym przetwarzanie dla innego niż założono problemu.

W przypadku wykorzystywania w systemie BASIC interpretatora, wszystkie trzy klasy błędów nakładają się na siebie. Do ich wyszukania należy więc też zatrudnić komputer.

10.1. Błędy składniowe i wykonywania

Oba rodzaje błędów są wykrywane bezpośrednio przez system BASIC. W opisywanej wersji języka z interpretatorem momentem wykrycia ewentualnego błędu składniowego w instrukcji jest pierwsze jej wykonanie. Jest więc możliwe, że błędy składniowe mogą się ujawnić dopiero w kolejnym przebiegu programu, gdyż wcześniej instrukcje z błędami nie były wykonywane.

Komunikat sygnalizujący błąd składniowy linii ma najczęściej postać

? Syntax error in line *numer_linii*

Numer_linii wskazuje na linię programu z instrukcją, w której dany błąd wystąpił.

Błąd składniowy znajduje się zawsze w linii, w której został on zasygnalizowany. Przeważnie jest to:

a) błędne zapisanie słowa kluczowego lub też pozostała część linii nie jest zgodna ze składnią instrukcji określonej pierwszym słowem kluczowym;

b) opuszczenie elementu składni instrukcji lub też błędne jego zapisanie – nieprawidłowy identyfikator zmiennej, zła postać stałej typu liczbowego lub tekstowego, brak operatora w wyrażeniu lub separatora na liście zmiennych czy liście wartości itp.
Może się również zdarzyć, że tekst linii jest zgodny ze składnią i nie widać żadnych odstępstw. W takim przypadku jest możliwe, że przypadkowo wprowadzono znak typu CTRL-X (inny niż sterujący), który z zasady nie jest widoczny w trakcie listowania tekstu programu.

Po komunikacie sygnalizującym błąd składniowy, system BASIC przechodzi na poziom edycji. Na poziomie tym możliwe jest poprawienie tekstu linii korzystając z własności edytora linii (analogicznie jak dla zlecenia EDIT – w tej książce opis funkcjonowania edytora został pominięty). Tutaj na razie zaleca się:

a) przejście na poziom wprowadzania poleceń (zlecenie edytora Q i następnie znak sterujący RETURN),

b) wylistowanie fragmentu programu z błędną linią za pomocą zlecenia LIST,

c) ewentualne sprawdzenie w trybie bezpośrednim wartości niektórych interesujących nas zmiennych,

d) wyszukanie w błędnej linii przyczyny błędu i powtórne wprowadzenie poprawnego tekstu linii z tym samym numerem linii.

Dla ilustracji rozważmy fragment sesji uruchamiania prostego programu.

```
Ok
10 INPUT a,b,c
20 LET  x=5*a$(b+)
30 PRINT x
40 END
RUN
? 5,7,3
Synta:x error in 20          'komunikat o bledzie w skladni,
20 LET  x=5*a$(b+)   ^0      'przejscie na poziom wprow. polecen,
PRINT  a;b;c;x               'sprawdzenie wartosci zmiennych,
 5   7   3   0
Ok
20 LET  x=5*a$(b+c)          'poprawiona linia,
RUN
? 5,7,3
 250
Ok
```

Na podobnej zasadzie są wykrywane błędy składniowe zlecenia – natychmiast po jego przekazaniu do wykonania. Zlecenie z błędem składniowym nie jest wykonywane, wystarczy więc tylko powtórnie wprowadzić zlecenie – już z poprawnie podanymi parametrami.

Błąd wykonywania jest wykrywany przez sprawdzenie zgodności argumentów aktualnie wykonywanej operacji z jej kontekstem w pozostałej części programu. Błędy te mogą wystąpić przy pierwszym lub kolejnym wykonywaniu tej samej instrukcji. Błąd jest sygnalizowany dla linii, w której został on wykryty. Jednak bezpośrednia przyczyna powstania takiego błędu jest często związana z błędnym lub niepoprawnym zapisem innych linii programu. Przeważnie jest to:

a) brak definicji funkcji, deklaracji tablicy lub instrukcji domykającej instrukcję cyklu czy powrotu z podprogramu, lub też brak linii o danym numerze;
b) niepoprawna, niezgodna z oczekiwaniem wartość zmiennej, prowadząca w tym kontekście do nieokreślonych wyników obliczeń;
c) wyczerpanie obszaru pamięci przeznaczonej na zapis tekstu programu, wartości zmiennych lub bieżące obliczenia.

Wystąpienie błędu wykonywania może też świadczyć o błędnej logice programu i wadach zaprojektowanego algorytmu.

Większość przyczyn błędów wykonywania programu omówiono w podrozdziałach „Sytuacje błędne". Tutaj ograniczymy się tylko do podsumowania tych rozważań i wyciągnięcia wniosków ułatwiających zapobieganie, wykrywanie oraz usuwanie przyczyn tego rodzaju błędów.

Pierwsza grupa błędów wykonywania jest spowodowana brakami w tekście programu. Korzystając z polecenia listowania należy przejrzeć fragment programu logicznie związany z linią z sygnalizowanym błędem. Przeważnie wystarczy tylko dopisać brakującą definicję, deklarację lub instrukcję z odpowiednim numerem linii. Przy okazji warto jednak sprawdzić, czy struktura sterowania tego fragmentu programu jest poprawna.

Omówienie drugiej grupy błędów wykonywania rozpoczniemy od przytoczenia fragmentu programu

```
10  DIM  x(20)
20  FOR i=0 TO 20
30    LET  x(i) = x(i+1)     '!!! dla i=20  blad indeksu !!!
40  NEXT i
```

W programie tym, błąd wykonywania zostanie wykryty dopiero przy 21 wykonaniu instrukcji z linii 30. Na podobnej zasadzie mogą powstawać błędy określoną jako próby dzielenia przez zero czy niepoprawną wartością argumentu spoza dziedziny obliczanej funkcji. Korzystając ze zleceń bezpośredniego wyprowadzania wartości zmiennych związanych z linią z błędem, przeważnie udaje się dość szybko określić, która to zmienna jest przyczyną błędu. Znacznie gorzej przedstawia się problem stwierdzenia, dlaczego akurat zmienna ta ma taką wartość, gdyż według naszego przekonania powinna mieć zupełnie inną – tę właściwą. Przyczyny przypisania zmiennej innej niż oczekiwana wartości mogą być spowodowane:

a) błędną postacią zapisu wyrażeń wyznaczających poprzednio wartość tej zmiennej;
b) wykorzystaniem zmiennej w innym miejscu programu do obliczeń lokalnych – na przykład w podprogramach;
c) pomyłką w identyfikatorze zmiennej, przy przypisywaniu jej uprzednio wartości, spowodowaną przestawieniem znaków lub wstawieniem innego podobnego w kształcie znaku;
d) nieprawidłowym określeniem wartości parametrów instrukcji cyklu lub określeniem warunku w instrukcji warunkowej powodującej wyznaczenie błędnej liczby iteracji, podczas których obliczana jest wartość zmiennej;
e) pominięciem w programie nadawania zmiennej wartości początkowej;
f) błędami w logice programu.

W języku BASIC zmienne proste są deklarowane niejawnie przez ich pierwsze wykorzystanie w wykonywanych instrukcjach. Stąd też większość omawianych tutaj przyczyn błędów jest z tym związana, gdyż wystąpienie w programie przypadkowego identyfikatora zmiennej nie jest sygnalizowane. Dodatkowo większość interpretatorów przypisuje każdej nowej zmiennej wartość równą zeru lub pustą (dla zmiennych typu tekstowego), co jest często wygodne, ale daje małe

szanse wykrycia takich przypadkowych zmiennych. Pozostaje więc jedynie dokładne przejrzenie tekstu programu i sprawdzenie, czy wszystkie występujące w nim zmienne są nam znane co do ich przeznaczenia. Negatywny wynik takich poszukiwań oraz błędy w logice programu lub algorytmu zmuszają nas do podjęcia bardziej metodycznych działań. Omówimy je dalej.

Po zatrzymaniu programu z sygnalizacją wyczerpania obszaru pamięci dokonujemy analizy zajętości pamięci przez program. Kolejno analizujemy:

a) zajętość pamięci przez tablice zadeklarowane jawnie oraz standardowo w programie, pamiętając o tym, że każda wartość typu liczbowego zajmuje przeważnie 5 bajtów, a każdy tekst liczbę bajtów równą długości tekstu;
b) liczbę wykorzystywanych w programie zmiennych;
c) zajętość pamięci przez tekst programu, zwracając szczególną uwagę na długości tekstów komentarzy, liczbę odstępów w zapisach tekstów linii;
d) stopień skomplikowania obliczania wyrażeń oraz zagnieżdżenie instrukcji cyklu i zagłębienie wywołań podprogramów.

Zmniejszenie wielkości programu można uzyskać dokonując kolejno:

a) zmniejszenia rozmiaru deklarowanych tablic – co jest stosunkowo proste, jeżeli odpowiednio napisaliśmy program parametryzując zakresy rozmiarów;
b) usunięcia zbyt długich komentarzy (tak, wiem, że dotychczas namawiałem do komentowania programu, a teraz trzeba to usunąć – wbrew pozorom nie ma w tym sprzeczności);
c) reorganizacji struktury programu przez wyodrębnienie wspólnych fragmentów w postaci sparametryzowanych podprogramów oraz jako definicji funkcji wyliczających wartości wyrażeń.

Podane tutaj metody nie wyczerpują wszystkich możliwości. Wykorzystanie ich wymaga jednak znacznie lepszej znajomości języka BASIC oraz zasad jego realizacji w danym systemie.

10.2. Błędy logiki programu

Programem poprawnym jest program, który dla prawidłowo podanego zestawu danych dokonuje przetworzenia ich w wyniki zgodnie z założeniami rozwiązywanego problemu, a dla każdego innego zestawu danych nie należącego do klasy rozwiązywanego problemu, sygnalizuje ich niepoprawność.

Weryfikacja poprawności programu wymaga przeanalizowania uzyskanych wyników pod kątem ich zgodności z założeniami realizowanego problemu. Stąd też elementem definicji problemu powinno być określenie przewidywanych wyników dla przykładowych zestawów danych. Oprócz przypadków trywialnych, dla których możemy porównać wyniki z wartościami uzyskanymi innymi meto-

dami, trudno jest podać, jakie wyniki są poprawne. Jeżeli bowiem przed napisaniem i wykonaniem programu znamy wyniki, to po co pisać program. Jest więc konieczne opracowanie *metod testowania* programu, które mogą ułatwić nam stwierdzenie jego poprawności.

Błąd logiki programu może mieć swoją przyczynę:

a) w błędnie zaprojektowanym algorytmie rozwiązania problemu nie uwzględniającym wszystkich możliwych warunków decydujących o kolejności przetwarzania danych i interpretacji wyników pośrednich,
b) w błędnie zakodowanym algorytmie w postaci programu pomijającym niektóre fragmenty algorytmu lub ograniczającym jego możliwości.

Ujawnienie błędów logiki programu może nastąpić częściowo w trakcie wykonywania programu poprzez sygnalizację błędów wykonywania. Jedyną metodą stwierdzenia braku błędów logiki jest pełne przetestowanie programu dla wszystkich możliwych rodzajów zestawów danych z uwzględnieniem przetestowania wszystkich ścieżek sterowania w programie. Dla dużych i skomplikowanych programów pełne ich przetestowanie jest bardzo czasochłonnym i kosztownym przedsięwzięciem.

Podsumowując dotychczasowe rozważania należy powiedzieć, że brak błędów składniowych i wykonywania w uruchamianym programie nie świadczy o jego poprawności. Stwierdzenie poprawności programu może nastąpić dopiero po wykonaniu na podstawie wyników testów, że funkcjonowanie programu jest zgodne z założeniami problemu, dla którego został on wykonany. Pracochłonność uzyskania takiego stwierdzenia często przekracza pracochłonność dotychczasowych działań, a więc zaprojektowania algorytmu, zakodowania go w języku programowania, wprowadzeniu programu do komputera oraz usunięcia błędów składniowych i wykonywania.

Do wykrywania błędów logiki algorytmu i programu istnieją różne metody oparte na tablicach decyzyjnych oraz definiowaniu niezmienników dla wykonywanych obliczeń. Omówienie tych metod wykracza poza ramy tej książki. Tutaj zajmiemy się tylko tymi elementami języka i systemu BASIC, które mogą wspomóc nas w prześledzeniu dróg sterowania w programie, określających kolejność wykonywania instrukcji programu.

Testowanie programu polega na jego wykonywaniu dla specjalnie opracowanych danych testujących umożliwiających zweryfikowanie poprawności poprzez:

a) sprawdzenie działania programu w warunkach granicznych lub dla osobliwych wartości danych,
b) oszacowanie dokładności obliczeń,
c) sprawdzenie odporności programu na błędy,
d) sprawdzenie wszystkich dróg sterowania.

Właściwe sprawdzenie poprawności programu wymaga opracowania zestawu testów umożliwiających zweryfikowanie wszystkich dróg sterowania. Przygotowanie testów polega na opracowaniu różnych zestawów danych, dla których wyniki uzyskane po wykonaniu programu możemy sprawdzić pod względem ich zgodności z założeniami rozwiązywanego problemu. Trudno jest podać uniwersalne zasady i kryteria doboru takich testów, gdyż jest to zależne od rodzaju problemu i zastosowanego algorytmu. Przeanalizujmy więc tylko podstawowe zasady, jakimi należy się kierować.

1. Test powinien sprawdzać działanie programu, w warunkach granicznych. Przykładowo więc należy z zakresu możliwych wartości danych wybierać wartości ekstremalne, a także równe zeru.
2. Kolejne testy w zestawie powinny się różnić tylko w niektórych wartościach dla sprawdzenia różnic w wartościach wyników. Na tej podstawie możliwe jest też oszacowanie dokładności, z jaką będą musiały być zadawane wartości danych dla uzyskania pożądanej dokładności wyników.
3. Jest pożądane, aby dla każdego zestawu danych testujących zostały oszacowane prawdopodobne wartości wyników, jakie powinny być uzyskane. Jednocześnie warto jest określić wartości wyników pośrednich, które dla zadanych danych testujących powinny być uzyskane w określonych punktach wykonywanego programu.
4. Należy ograniczać czas wykonywania programu dla danych testujących poprzez minimalizowanie liczby iteracji, liczby przetwarzanych wartości, wielkości rozmiarów analizowanych tablic czy długości przetwarzanych napisów.
5. Część testów powinna zawierać wartości danych niezgodne z założeniami rozwiązywanego problemu. W tym przypadku wykonanie programu powinno spowodować wyprowadzenie odpowiednich komunikatów sygnalizujących odporność programu na błędy.
6. Pełny zestaw testów powinien weryfikować wszystkie istniejące drogi sterowania w programie. Stąd też przygotowując testy należy przeanalizować sieć działań zaznaczając w niej, który z testów weryfikuje daną drogę w sieci.

10.3. Technika uruchamiania programu

W procesie uruchamiania programu wyróżniamy dwa etapy:
1. Usuwanie z programu błędów składniowych i błędów wykonywania.
2. Testowanie programu z usuwaniem błędów logicznych programu.

Usuwanie błędów z programu polega na uruchomieniu programu z bardzo prostymi zestawami danych w celu sprawdzenia poprawnego zapisania tekstu programu. Wykorzystywane zestawy danych powinny umożliwiać sprawdzenie praktycznie każdej drogi sterowania w programie. Uzyskiwane wyniki programu tymczasem nas nie interesują.

Dla pierwszego etapu uruchamiania programu warto jest przygotować testujący zestaw danych, jako strumień danych wewnętrznych, aby nie trzeba było go wielokrotnie wprowadzać dla kolejnych prób uruchomienia programu. Jednocześnie staramy się minimalizować liczbę tych danych. Przykładowo więc:
1. Jeżeli nasz program jest przeznaczony do sortowania wektorów n-elementowych, to w zestawie danych ustalamy n równe 4 lub 5 zależnie od przyjętej metody sortowania. Na podobnej zasadzie zmniejszamy liczność próbki, dla której program dokonuje analizy statystycznej.
2. Jeżeli nasz program przetwarza tablice, to w parametrach programu zmniejszamy rozmiary tych tablic do niezbędnego minimum (istnieje duża różnica we wprowadzeniu tablicy 12-elementowej (3,2) i tablicy 144-elementowej (11,11)).
3. Jeżeli nasz program przetwarza teksty, to zmniejszamy ich długość, tak aby nie trzeba było wprowadzać zbyt długich napisów.

Minimalizujemy również czas działania programu.
1. Przeważnie zmniejszenie liczby danych pociąga za sobą zmniejszenie czasu działania programu.
2. Jeżeli nasz program wykonuje obliczenia iteracyjne z zadaną dokładnością, wartość epsilon ustalamy jako niezbyt małą.
3. Największe korzyści w minimalizacji czasu działania programu uzyskamy zmniejszając wszędzie liczbę wykonań instrukcji cyklu oraz liczbę iteracji.

W tym momencie ujawniają się zalety pisania programów według kanonów dobrego stylu programowania. Uwzględnienie możliwości parametryzacji programu na etapie jego projektowania teraz zwraca się z nawiązką. Z drugiej strony czytelność i łatwość modyfikacji tekstu programu daje nam szansę prostego przystosowania programu do częstych i krótkich w działaniu wykonań.

Po wstępnym przystosowaniu programu do akceptacji prostych zestawów danych przystępujemy do uruchamiania programu. W rozpatrywanym systemie BASIC wystąpienie błędu przerywa dalsze wykonywanie programu. Po poprawieniu błędu składniowego możemy wznowić wykonywanie programu począwszy od błędnej instrukcji zleceniem

GOTO *numer_linii.*

Działanie takie (niezgodne z zasadami) ma sens jedynie w niektórych warunkach, gdy poprawiając linię nie naruszono wewnętrznej struktury programu i obszarów przeznaczonych na zmienne. Najczęściej jednak należy po prostu dla tego samego zestawu danych uruchomić program od początku zleceniem

RUN

Warto jednak przy okazji przed uruchomieniem, a nawet przed poprawieniem linii, sprawdzić w trybie bezpośrednim wartości niektórych zmiennych:

a) przechowujących dane programu – analizując ich zmiany w stosunku do wartości z zestawu danych;
b) sterujących w instrukcji cyklu, jeżeli ostatnio wykryty błąd był w linii w niej zawartej;
c) ostatnio obliczanych w poprzednich instrukcjach.

Uzyskane wartości należy przeanalizować pod kątem ich zgodności z oczekiwaniami, a nawet warto gdzieś zanotować.

Analogicznie postępujemy po wykryciu błędu wykonywania, jeżeli od razu potrafimy ten błąd poprawić. Nieco trudniejsza jest sytuacja, gdy nie możemy znaleźć przyczyny błędu.

Po usunięciu błędów składniowych i wykonywania przechodzimy do drugiego etapu uruchomienia programu – testowania w celu zweryfikowania jego poprawności. W naszym przypadku etap ten polega na wykryciu ewentualnych błędów logicznych występujących w programie. Te same metody stosujemy też w przypadku występowania błędu wykonywania, dla którego nie możemy zlokalizować przyczyny jego powstawania. Przed rozpoczęciem tego etapu uruchamiania musimy dokonać *instrumentacji programu.*

Instrumentacja programu polega na świadomym wprowadzeniu do tekstu programu instrukcji umożliwiających stałe kontrolowanie jego działania. Elementami instrumentacji programu są:
a) wprowadzenie do tekstu programu wydruków wartości pośrednich,
b) wprowadzenie do tekstu programu instrukcji wstrzymywania STOP,
c) wykorzystanie instrukcji śledzenia działania programu.

Do wyprowadzania wartości pośrednich wykorzystujemy instrukcję wyprowadzania PRINT, ewentualnie umieszczone w instrukcjach warunkowych.

```
127 IF  ts=1  THEN PRINT "127" ; w1;w2;w3
```

Instrukcje te wstawiamy w newralgiczne z punktu widzenia logiki programu miejsca, opisując wyprowadzone wartości charakterystycznym oznaczeniem (np. numerem linii lub symbolem). Najczęściej miejscami tymi są: linie poprzedzające oraz następujące po wywołaniu podprogramu, zakończenie poszczególnych modułów, czy zakończenie wprowadzania danych. Instrukcja warunkowa z relacją typu $ts = 1$ umożliwia nam okresowe żądania tych wydruków lub ich zaniechanie ($ts = 0$). Jednocześnie musimy również pamiętać, że nadmiar tych dodatkowych wartości może znacznie ograniczyć nas w zrozumieniu tego, co się dzieje w wykonywanym programie. Stąd też na przykład wewnątrz wielokrotnie wykonywanych instrukcji cyklu lepiej jest sygnalizować każdy cykl pojedynczym znakiem, jak np.

```
100  FOR i=1 TO n
101    PRINT ".";
110    '...
120    '...
900  NEXT i
```

Wydruki pośrednie pozwalają na szybkie zlokalizowanie nieprawidłowości dotyczących obliczeń wartości zmiennych.

Po uruchomieniu programu, tracimy nad nim kontrolę i jedynie korzystając z funkcji przerwania wywoływanej znakiem CTRL-C możemy przerwać jego wykonywanie w przypadkowych miejscach. Dla świadomej kontroli przebiegu drogi sterowania w wykonywanym programie wprowadzamy również w newralgiczne jego miejsca instrukcję wstrzymania STOP. W tym przypadku po wstrzymaniu działania programu możemy w trybie bezpośrednim sprawdzić wartości wybranych zmiennych i następnie wznowić wykonywanie programu poleceniem kontynuacji CONT. W ten sposób jest możliwe prześledzenie działania programu i ewentualne wykrycie błędów w logice sterowania. Musimy przy tym pamiętać, że poprawienie tekstu programu po wstrzymaniu wymaga uruchomienia jego od początku zleceniem RUN.

Stałe śledzenie kolejności wykonywanych instrukcji programu możemy uzyskać, korzystając z własności *polecenia włączania i wyłączania śledzenia.*

TRON
TROFF

Polecenie włączenia śledzenia programu TRON powoduje dla aktualnie wykonywanego programu wyprowadzanie w nawiasach kwadratowych numeru linii każdej rozpoczętej instrukcji. Polecenie to jest realizowane do chwili wprowadzenia polecenia wyłączenia śledzenia TROFF lub zerowania systemu poprzez jego wyłączenie i włączenie, znakiem sterującym zerowanie lub poleceniem NEW. Oba polecenia mogą być podawane jako zlecenia w trybie bezpośrednim lub jako instrukcje programu.

Wykorzystując odpowiednio instrukcje włączania i wyłączania śledzenia możemy uzyskać ślad wykonywania podstawowych fragmentów programu. Musimy sobie przy tym zdawać sprawę, że włączenie śledzenia dla instrukcji cyklu wykonującej się na przykład 1000 razy spowoduje wyprowadzenie $(n+1)*1000+1$ numerów linii, gdzie n jest liczbą linii zawartych w cyklu.

Ilustracją funkcjonowania polecenia śledzenia programu są dwa przykłady.

```
10 LET sp=0
20 LET sm=0
30 READ  n
40 FOR i=1 TO n
50   READ x
60   IF  x>0  THEN  90
70     LET  sm=sm+1
80     GOTO 100
90   LET  sp=sp+1
100 NEXT i
110 PRINT  sm,sp
120 END
130  DATA 5
140 DATA  1,-5,-4,-6,5
150  END
TRON
Ok
RUN
[10][20][30][40][50][60][90][100][50][60][70][80][
100][50][60][70][80][100][50][60][70][80][100][50]
[60][90][100][110] 3                2
[120]
Ok
TROFF
Ok


LIST
10 LET sp=0
20 LET sm=0
30 READ  n
40 FOR i=1 TO n
45                  TRON
50   READ x
60   IF  x>0  THEN  90
70     LET  sm=sm+1
80     GOTO 95
90   LET  sp=sp+1
95                  TROFF
96                  PRINT  x
100 NEXT i
110 PRINT  sm,sp
120 END
130  DATA 5
140 DATA  1,-5,-4,-6,5
150  END
Ok
RUN
[50][60][90][95] 1
[50][60][70][80][95]-5
[50][60][70][80][95]-4
[50][60][70][80][95]-6
[50][60][90][95] 5
 3              2
Ok
```

Wykorzystując wszystkie trzy, lub tylko niektóre, metody instrumentacji programu oraz odpowiednio przygotowane zestawy danych testujących kolejno weryfikujemy poprawność fragmentów programu.

Działania te wymagają dużo cierpliwości, ale prowadzone metodycznie przybliżają nas do stwierdzenia, że program jest prawie poprawny. Jak uczy doświadczenie, stuprocentowej pewności prawie nigdy mieć nie możemy, gdyż nawet po wielokrotnej eksploatacji programu możemy nagle stwirerdzić istnienie w nim błędu.

Stąd też, po rozpoczęciu eksploatacji programu, oczywiście po usunięciu z tekstu programu elementów jego instrumentacji, zachowujemy na wszelki wypadek wersję zinstrumentalizowaną. Proszę wierzyć, że przyjdzie taki moment, że stanie się znowu przydatna.

Pozostaje jeszcze odpowiedzieć na pytanie, co robić, jeżeli pomimo naszych usilnych starań program działa błędnie. No cóż – proponuję kolejno następujące działania:

1. Uzbroić się w cierpliwość i jeszcze raz spokojnie przejrzeć tekst programu, najlepiej wspólnie z innym programistą, wyjaśniając mu przeznaczenie każdej instrukcji przez nas napisanej. Często bowiem tylko nasze przeświadczenie o określonym działaniu fragmentu programu było poprawne – w rzeczywistości jest zupełnie inaczej.
2. Na podstawie tekstu programu należy odtworzyć sieć działań porównując ją następnie z wersją przez nas na początku skonstruowaną dla danego algorytmu. Porównanie to często ujawnia znaczne rozbieżności będące z reguły przyczyną poszukiwanego przez nas błędu.
3. Nie wstydzić się i poprosić o pomoc bardziej doświadczonych programistów. Wielu z nich bowiem spotykało się już z takim problemem i często mogą prawie natychmiast wskazać potencjalne miejsca programu, w których znajduje się przyczyna błędu.
4. Napisać jeszcze raz porządnie program od początku, korzystając ze zdobytych już doświadczeń i stosując wszystkie kanony dobrego stylu programowania. W tym miejscu potwierdza się konieczność pisania ładnych i eleganckich programów.
5. I wreszcie, jeżeli żadna z podanych wyżej rad nie przyniesie pozytywnego rezultatu, można zacząć podejrzewać używany przez nas komputer oraz wykorzystywane oprogramowanie o błędne funkcjonowanie. Określenie zaś przyczyny jego uszkodzenia niestety przekracza ramy tej pracy. Konieczna jest więc fachowa diagnoza i naprawa. Dla dobrego sprzętu sytuacja taka zdarza się dość rzadko (dla złego – niestety za często).

Styl programowania

Podczas uruchamiania programu ujawniają się przede wszystkim zalety stosowania kanonów dobrego stylu programowania. Konieczność zweryfikowania i powtórnego przeanalizowania prawie każdej instrukcji czy fragmentu programu jest dobrym sprawdzianem, czy napisaliśmy program, którego działanie możemy skontrolować.

Tutaj możemy jeszcze uwzględnić dodatkowe wymagania związane z samym procesem uruchamiania programu.

Instrumentacja programu. Przystosowanie tekstu programu do procesu kontrolowanego uruchamiania powinno się odbywać już w trakcie jego opracowywania. Jest bowiem znacznie prościej od razu umieszczać w programie instrukcje wyprowadzania wyników pośrednich czy wstrzymywanie, niż dopiero w trakcie poszukiwania błędów. Oczywiście instrukcje te powinny być odpowiednio oznaczone, aby łatwe było ich późniejsze usunięcie z gotowego programu. Jako zasadę należy przyjąć umieszczanie tych instrukcji szczególnie:

a) po fragmencie programu wprowadzającym wartości danych, tak aby być pewnym, że pozostała część programu będzie przetwarzać wartości dane zgodnie z założeniami;

b) po każdym logicznie wyodrębnionym module, szczególnie gdy wyliczone przez niego wartości mają istotny wpływ na przebieg pozostałej części programu;

c) w tych fragmentach, co do których nie jesteśmy pewni, czy prawidłowo opracowaliśmy algorytm obliczeń.

Z drugiej strony musimy pamiętać, że nadmiar dodatkowych wyprowadzeń wartości zmiennych zaciemnia obraz działania programu. Stąd też każde takie wyprowadzenie wartości pośrednich powinno być łatwo identyfikowane oraz część z nich możemy na przykład umieścić jako komentarze do ewentualnego wykorzystania, jeżeli będą potrzebne.

```
100    REM  PRINT  "alfa"; wl; wx; k; a
110 '....
120 '....
200    REM  PRINT "koniec przeszukiwania tablicy"
210    REM  STOP
```

Modularność programu. Wprowadzając pojęcie podprogramu zwracałem uwagę na konstruowanie struktury programu jako zbioru dobrze zdefiniowanych modułów. Zasada ta, niezwykle istotna dla zachowania dobrego stylu programowania, ma dodatkową zaletę w procesie uruchamiania. Zamiast bowiem uruchamiać cały skomplikowany program, możemy rozłącznie sprawdzać działanie poszczególnych jego modułów. Oczywiście w takim przypadku konieczne jest dopisanie dla każdego takiego modułu dodatkowych fragmentów wprowadzają-

cych dla niego wartości danych testujących i wyprowadzających wyniki. Postępowanie takie znacznie upraszcza postać danych testujących oraz przyspiesza i ułatwia proces uruchamiania programu. Dodatkowe pomocnicze moduły testujące możemy umieścić na końcu programu – wiążąc je z testowanym modułem instrukcjami przejścia GOTO lub wywoływania jako podprogramu.

Let's learn BASIC English

debugging odpluskwianie – potocznie używane określenie procesu usuwania błędów z programu
debugger program odpluskwiający, program wspomagający uruchomienie programu (w niektórych systemach BASIC istnieją polecenia umożliwiające analizowanie zmian wartości zmiennych)
trace ślad (tutaj ślad wykonywanych instrukcji jako ciąg ich numerów linii)
tracing śledzenie
tracer program śledzący (w BASICu jest to wbudowane w system polecenie)
TRON(TRacing ON) WŁĄCZenie ŚLEDZenia
TROFF(TRacing OFF) WYŁĄCZenie ŚLEDZenia
error błąd
fault uszkodzenie

END of lesson

Dodatki

A. Wartości kodów ASCII

znak	kod	znak	kod	znak	kod	znak	kod
NUL	0	␣	32	@	64	`	96
CTRL-A	1	!	33	A	65	a	97
STA	2	"	34	B	66	b	98
CTRL-C	3	#	35	C	67	c	99
EOT	4	$	36	D	68	d	100
ENQ	5	%	37	E	69	e	101
ACK	6	&	38	F	70	f	102
CTRL-G	7	'	39	G	71	g	103
CTRL-H	8	(	40	H	72	h	104
CTRL-I	9	)	41	I	73	i	105
LINEFEED	10	*	42	J	74	j	106
HOME	11	+	43	K	75	k	107
FORMFEED	12	,	44	L	76	l	108
RETURN	13	-	45	M	77	m	109
SO	14	.	46	N	78	n	110
CTRL-O	15	/	47	O	79	o	111
DLE	16	0	48	P	80	p	112
CTRL-Q	17	1	49	Q	81	q	113
CTRL-R	18	2	50	R	82	r	114
CRTL-S	19	3	51	S	83	s	115
DC4	20	4	52	T	84	t	116
CTRL-U	21	5	53	U	85	u	117
SYN	22	6	54	V	86	v	118
ETB	23	7	55	W	87	w	119
CAN	24	8	56	X	88	x	120
EM	25	9	57	Y	89	y	121
SUB	26	:	58	Z	90	z	122
ESCAPE	27	;	59	[	91	{	123
→	28	<	60	\	92	|	124
kursor ←	29	=	61	]	93	}	125
kursor ↑	30	>	62	^	94	~	126
kursor ↓	31	?	63	_	95	DEL	127

B. Wykaz komunikatów błędów

W wykazie umieszczono wszystkie możliwe dla pełnej standardowej wersji języka Microsoft BASIC komunikaty błędów. Liczby w nawiasach podają numer strony w książce, na której omówiono szerzej dany rodzaj błędu.

Bad file number
: Próba zapisu lub odczytu dla nieotwartego pliku.

Bad file mode
: Próba wykorzystania pliku w innym trybie niż został otwarty.

Bad file name
: Nieprawidłowa nazwa pliku – niezgodna z zasadami ustalonymi dla danego systemu.

Can't continue
: Próba wznowienia działania programu, który został zatrzymany z powodu błędu lub został podczas okresu wstrzymania zmodyfikowany lub po prostu już nie istnieje w pamięci RAM.

Direct statement in file
: Podczas ładowania programu wykryto zlecenie – wykonanie polecenia LOAD kończy się.

Disk I/0 error
: Awaria dysku.

Disk full
: Cała przestrzeń nośnika dysku jest zajęta.

Division by zero (91)
: Wykryto dzielenie przez zero.

File already open
: Próba otwarcia lub usunięcia pliku już otwartego.

File not found
: Odwołanie się do nieistniejącego aktualnie pliku.

FOR without NEXT (122)
W trakcie wykonywania instrukcji FOR nie znaleziono odpowiadającej jej instrukcji NEXT.

Illegal direct
Podane polecenie nie może być wykonane jako zlecenie.

Illegal function call (91, 213)
W wywołaniu funkcji standardowej wykryto nieprawidłową wartość lub typ parametru.

Input past end
Próba odczytu kolejnej wartości po odczytaniu z pliku wartości ostatniej.

Internal error
Awaria systemu.

Line buffer overflow
Próba wpisania z klawiatury zbyt wielu znaków w jednej linii.

Missing operand (91)
Wyrażenie zawiera argumenty bez operatora.

NEXT without FOR (122)
Zmienna sterująca w instrukcji NEXT nie występuje w odpowiadającej jej instrukcji FOR.

No RESUME
W sekwencji obsługi sytuacji wyjątkowych nie istnieje instrukcja RESUME.

Out of data (75)
Wyczerpanie wartości w strumieniu danych wewnętrznych.

Out of memory
Program nie mieści się w pamięci lub w trakcie jego wykonywania wystąpiło za duże zagłębienie instrukcji FOR...NEXT lub wywołań podprogramów GOSUB.

Out of string space (187)
Dla zmiennych tekstowych zabrakło miejsca w wydzielonym dla nich obszarze pamięci.

Overflow (91)
Wynik obliczeń przekracza zakres możliwej do zapamiętania w komputerze wartości.

Redimensioned array (154)
Próba powtórnej deklaracji tej samej tablicy.

Redo from start, Retype line (74)
Nieprawidłowy strumień danych dla instrukcji wprowadzania INPUT. Należy powtórzyć wprowadzenie danych.

RETURN without GOSUB (213)
Próba wykonania instrukcji RETURN przy braku śladu wywołania podprogramu.

RESUME without error
Próba wykonania instrukcji RESUME w innym niż zakończenie obsługi sytuacji wyjątkowej, kontekście.

String formula too complex (187)
Wyrażenie tekstowe jest zbyt skomplikowane.

Syntax error (52, 91)
W linii wykryto błąd składniowy polegający na opuszczeniu, zamianie, pomyłce itp. w zapisie elementów polecenia.

Subscript out of range (153)
Zmienna tablicowa odwołuje się do nieistniejącego elementu tablicy lub do tablicy o innej liczbie wymiarów.

Too many files
Próba otwarcia lub utworzenia nowego pliku w przypadku, gdy w katalogu istnieje już ich 255.

Type mismatch (75)
Niezgodność typów w przypadku wykorzystywania zmiennych liczbowych w miejscu zmiennych tekstowych lub odwrotnie.

Undefined line (52, 122)
Podany w poleceniu numer linii nie istnieje w programie.

Undefined user function (213)
Brak definicji funkcji.

Unprintable error
Dla danej wartości błędu nie istnieje komunikat go opisujący.

C. Microsoft BASIC-80

Omówienie zapisu:

XYZ	zarezerwowane słowa kluczowe języka BASIC.
[*fragment*]	fragment mogący występować opcjonalnie.
[*fragment*]...	fragment mogący występować zero, jeden lub więcej razy.

Skróty:

x	*wyrażenie_arytmetyczne,*	*x$,y$*	*wyrażenie_tekstowe*
n,m	*wyrażenie_całkowite,*	*np*	*numer_pliku*

Edycja programu

Polecenia

AUTO [*numer_linii* [, *inkrement*]]

Automatyczna numeracja linii z podanym *inkrementem* (standardowo przyjmuje się AUTO 10, 10).

DELETE [*numer_linii*][_*numer_linii*]

Usunięcie linii z programu w podanym zakresie, lub pojedynczej linii.

EDIT *numer_linii*

Edycja wskazanej linii.

LIST [*numer_linii* [_[*numer_linii*]]]

LLIST [*numer_linii* [_[*numer_linii*]]]

Wyświetlanie lub drukowanie fragmentu tekstu programu o podanym zakresie.

NEW

Wymazanie bieżącej zawartości pamięci RAM.

RENUM [[*nowy_numer*][,[*stary_numer*][,*inkr*]]]

Przenumerowanie linii programu z nową numeracją wraz ze zmianą numerów linii w instrukcjach.

Uruchomienie i śledzenie programu

Polecenia

RUN [*numer_linii*]

Uruchomienie wykonywania programu od początku lub wskazanej *numerem_linii* programu.

CONT

Wznowienie wykonywania programu od następnej po wstrzymanej instrukcji.

TRON

TROFF

Włączenie oraz wyłączenie (TROFF) śledzenia kolejno wykonywanych instrukcji programu.

Instrukcje programu

numer_linii instrukcja [:*instrukcja*]...

W jednej linii może się znajdować wiele *instrukcji* oddzielonych od siebie znakiem dwukropka.

REM *komentarz*

' *komentarz*

Zaznaczenie dowolnego *komentarza* opisującego program. Musi być ostatnią instrukcją w linii programu.

Definicje i deklaracje

litera [*litera_lub_cyfra*]...

ident – identyfikator bazowy zmiennej lub funkcji składający się maksymalnie z 40 znaków (nie może być słowem kluczowym!).
Nazwa zmiennej lub funkcji typu:

ident ident!	– rzeczywistego pojedynczej precyzji,
ident!	– rzeczywistego podwójnej precyzji,
ident%	– całkowitego,
ident$	– tekstowego.

Funkcje

FN*nazwa* [(*argument* [,*argument*]...)]

Wywołanie funkcji zdefiniowanej za pomocą instrukcji DEF FN... o typie wartości określonej ostatnim znakiem *nazwy.*

FRE(0)

FRE(*x$*)

Zwraca liczbę bajtów pamięci nie wykorzystywanych przez system BASIC lub liczbę wolnych bajtów przeznaczonych na teksty (FRE(*x$*)).

Instrukcje

DEFINT *litera* [– *litera*][,*litera*]...
DEFSNG
DEFDBL
DEFSTR

Definiowanie typów nazw identyfikatorów poprzez pierwszą *literę*.

DEF FN*nazwa* [(*parametr* [,*parametr*]...)] = *wyrażenie*

Definiowanie dla zadanych *parametrów* funkcji o wartości określonej typem *nazwy*.

CLEAR [, *wyrażenie_całkowite*]

Wyzerowanie wartości zmiennych i ustalenie górnego zakresu obszaru pamięci przeznaczonego na wykonywanie programu w BASICu.

DIM *nazwa_tablicy* (*rozmiar* [,*rozmiar*]...)

Deklarowanie tablicy o określonej liczbie wymiarów i *rozmiarze* w każdym wymiarze.

OPTION BASE *n*

Ustalenie dolnej granicy rozmiaru tablicy na wartość *n* (0 lub 1).

ERASE *nazwa_tablicy* [,*nazwa_tablicy*]...

Wymazanie tablicy z programu.

Obliczenia arytmetyczne

Operatory

arytmetyczne		*relacji*		*logiczne*	
↑	potęgowanie	=	równość	NOT	negacja
–	zmiana znaku	< >	nierówność	AND	iloczyn
*	mnożenie	<	mniejsze	OR	suma
/	dzielenie	>	większe	XOR	suma modulo2
MOD	modulo	>=	nie mniejsze	IMP	implikacja
\	dzielenie całkowite	<=	nie większe	EQV	równoważność
+	plus				
–	minus				

Funkcje arytmetyczne:

ABS(*x*)	Wartość bezwzględna.
SGN(*x*)	Znak wartości wyrażenia.
FIX(*x*)	Obcięcie do wartości całkowitej.
INT(*x*)	Obcięcie do wartości całkowitej nie większej od *x*.
CDBL(*x*)	Konwersja do wartości podwójnej precyzji.
CINT(*x*)	Konwersja do wartości całkowitej z zaokrągleniem.
CSNG(*x*)	Konwersja do wartości o pojedynczej precyzji.

SIN(x) Sinus x (x – w radianach).
COS(x) Cosinus x (x – w radianach).
TAN(x) Tangens x (x – w radianach).
ATN(x) Arctangens x (wynik w radianach).
EXP(x) Exponent x.
LOG(x) Logarytm naturalny x ($x > 0$).
RND(x) Zwraca liczbę losową z przedziału 0...1, zależnie od wartości x. Brak x lub $x > 0$ generuje kolejną liczbę z sekwencji, $x = 0$ powtarza poprzednio wygenerowaną, a $x < 0$ rozpoczyna od początku.

Instrukcje

LET *zmienna_arytmetyczna = wyrażenie_arytmetyczne*
Przypisanie wartości wyrażenia zmiennej.
SWAP *zmienna_arytmetyczna 1, zmienna_arytmetyczna 2*
Zamiana wartości w zmiennych
RANDOMIZE [*wyrażenie_arytmetyczne*]
Ustawienie wartości początkowej generatora liczb pseudolosowych według wartości wyrażenia lub konwersacyjnie z klawiatury.

Przetwarzanie tekstów

Operator

\+ konkatenacji (złożenia tekstów).

Funkcje tekstowe

ASC($x$$)
Zwraca wartość liczbową kodu pierwszego znaku z tekstu $x$$.
VAL($x$$)
Zwraca wartość liczbową zapisaną tekstowo w $x$$.
LEN($x$$)
Zwraca długość tekstu $x$$.
INSTR([n,]$x$$,$y$$)
Wyszukanie w tekście $x$$, począwszy od n-tego (lub pierwszego) znaku miejsca zawierania się tekstu $y$$. W wyniku otrzymujemy wartość pozycji początku tekstu lub wartość 0, gdy tekst $y$$ nie istnieje w $x$$.
CHR$($n$)
Tworzy jednoznakowy tekst dla kodu n.
OCT$($x$)
Tworzy tekst reprezentujący wartość wyrażenia x w postaci oktalnej.
HEX$($x$)
Tworzy tekst reprezentujący wartość wyrażenia x w postaci szesnastkowej.

STR$(*x*)
: Tworzy tekst reprezentujący wartość wyrażenia *x* w postaci dziesiętnej.

LEFT$(*x$,n*)
: Zwraca *n* pierwszych znaków z tekstu *x$*.

RIGHT$(*x$,n*)
: Zwraca *n* ostatnich znaków z tekstu *x$*.

MID$(*x$,n*[,*m*])
: Zwraca *m* (lub 1) znaków ze środka tekstu *x$* począwszy od znaku *n*.

SPACE$(*n*)
: Tworzy tekst złożony z *n* spacji.

STRING$(*n*,*m*)
STRING$(*n*,*x$*)
: Tworzy tekst złożony z *n* jednakowych znaków o kodzie *m*, lub pobranego jako pierwszy znak z tekstu *x$*.

Instrukcje

LET *zmienna_tekstowa* = *wyrażenia_tekstowe*
: Przypisanie wartości wyrażenia tekstowego zmiennej tekstowej.

MID$(*zmienna_tekstowa,n*[,*m*]) = *wyrażenie_tekstowe*
: Wymiana w tekście zawartym w *zmiennej_tekstowej*, począwszy od znaku z pozycji *n*, fragmentu na tekst dany *wyrażeniem_tekstowym o długości m lub długości tekstu z wyrażenia.*

SWAP *zmienna_tekstowa 1, zmienna_tekstowa 2*
: Wymiana zawartości w zmiennych tekstowych.

Wprowadzanie i wyprowadzanie wartości

Funkcje wprowadzania i redagowania tekstu

INKEY$	Udostępnia pojedynczy znak jako wartość tekstową z bufora klawiatury.
INPUT$(*n*)	Zwraca *n*-elementowy tekst złożony z kolejnych znaków wprowadzonych z klawiatury.
TAB(*n*)	Powoduje przesunięcie wskaźnika wyprowadzania wartości do bufora na *n*-tą pozycję.
SPC(*n*)	Powoduje wyprowadzenie do bufora *n* spacji.
POS(*n*)	Udostępnia aktualną wartość wskaźnika pozycji
LPOS(*n*)	wyprowadzanego znaku na ekran lub papier drukarki. Wartość *n* jest nieistotna.

Instrukcje

DATA *stała* [,*stała*]...
: Tworzy strumień danych wewnętrznych.

READ *zmienna* [,*zmienna*]...
Przypisuje zmiennym z listy wprowadzania wartości ze strumienia danych wewnętrznych.

RESTORE [*numer_linii*]
Ustala wskaźnik wprowadzania wartości w strumieniu danych wewnętrznych według *numeru_linii* instrukcji DATA lub na początek strumienia.

INPUT[;]["*tekst_zachęty*";] *zmienna* [,*zmienna*]...
Przypisuje zmiennym wartości wprowadzane z klawiatury, wyprowadzając ewentualnie *tekst_zachęty* oraz zawsze znak zapytania. Znak średnika po słowie INPUT blokuje zmianę wiersza po wprowadzeniu sekwencji wartości.

LINE INPUT [;]["*tekst_zachęty*";] *zmienna_tekstowa*
Przypisuje *zmiennej_tekstowej* tekst będący pełną linią wprowadzonych z klawiatury znaków.

PRINT [USING "*wzorzec*";] *wyrażenie* [*sep wyrażenie*]...[*sep*]
LPRINT [USING "*wzorzec*";] *wyrażenie* [*sep wyrażenie*]...[*sep*]
Wyprowadza wartości wyrażeń tekstowych oraz arytmetycznych według wzorca lub według standardu modyfikowanego rodzajem separatora , lub ; . Separator na końcu listy wyprowadzania blokuje zmianę wiersza. Pusta lista wyprowadzania oznacza wyprowadzenie wiersza pustego. Zamiast słowa kluczowego PRINT można używać znaku zapytania? oraz LPRINT do drukowania na papierze.

WRITE *wyrażenie* [,*wyrażenie*]...
Wyprowadza wartości wyrażeń oddzielając je przecinkami oraz ujmując wartości tekstowe w cudzysłowy.

WIDTH LPRINT *wyrażenie_całkowite*
Określa szerokość wyprowadzanego wiersza na ekranie lub na papierze drukarki.

Sterowanie programem

Instrukcje

END
Zakończenie wykonywania programu.

STOP
Wstrzymanie wykonywania programu — możliwe jego wznowienie poleceniem CONT.

GOTO *numer_linii*
Bezwarunkowe przejście do linii o podanym *numerze_linii*.

IF *wyrażenie_warunkowe* THEN *numer_linii* [ELSE *numer_linii*]
IF *wyrażenie_warunkowe* GOTO *numer_linii* [ELSE *numer_linii*]

IF *wyrażenie_warunkowe* THEN instrukcje [ELSE instrukcje]

Warunkowe wykonanie przejścia do linii podanej po słowie THEN, jeżeli warunek jest spełniony (czyli, gdy wartość wyrażenia nie jest równa 0), a w przeciwnym razie do linii podanej po słowie ELSE lub bezpośrednio następnej. Podobnie możliwe jest wykonanie odpowiedniej sekwencji instrukcji.

FOR *zm_sterująca* = *wart_pocz* TO *wart_konc* [STEP *krok*]

Zdefiniowanie parametrów cyklu dla wykonywania kolejnych instrukcji dla wartości zmiennej sterującej z przedziału określonego wartością początkową i końcową z podaną wartością kroku lub standardowo co 1.

NEXT [*zm_sterująca* [,*zm_sterująca*]...]

Domknięcie sekwencji instrukcji wykonywanych cyklicznie i powrót do wyznaczenia kolejnej wartości zmiennej sterującej.

WHILE *wyrażenie_warunkowe*

Określenie początku cyklicznie wykonywanych sekwencji instrukcji póki wartość wyrażenia warunkowego jest prawdą.

WEND

Domknięcie sekwencji instrukcji wykonywanych cyklicznie dla spełnionego wyrażenia warunkowego. Niespełnienie wartości wyrażenia w odpowiadającej instrukcji WHILE powoduje przejście do instrukcji z linii następnej po WEND.

GOSUB *numer_linii*

Przejście do wykonywania sekwencji instrukcji począwszy od podanego *numeru_linii* z zapamiętaniem śladu.

RETURN

Powrót z wykonywania sekwencji instrukcji do następnej według śladu wywołania podprogramu.

ON *wyrażenie_całkowite* GOTO *numer_linii* [,*numer_linii*]...

Przejście do wykonywania instrukcji z linii wybranej z listy numerów linii według wartości *wyrażenia_całkowitego.*

ON *wyrażenie_całkowite* GOSUB *numer_linii* [,*numer_linii*]...

Przejście do wykonywania sekwencji podprogramu instrukcji począwszy od linii wybranej z listy numerów linii według wartości całkowitej wyrażenia.

Współpraca z plikami

Funkcje wspomagające

EOF(*np*)

Zwraca wartość równą −1 (prawda), jeżeli został osiągnięty koniec pliku sekwencyjnego.

LOC(*np*)
Dla plików o dostępie bezpośrednim zwraca numer rekordu ostatnio czytanego lub zapisywanego lub wartość 0, jeżeli jeszcze nie była wykonywana operacja na pliku.

CVI (*napis_2_bajtowy*)
CVS (*napis_4_bajtowy*)
CVD (*napis_8_bajtowy*)
Dokonuje konwersji wartości tekstowej odczytywanej z pliku o dostępie bezpośrednim w wartość liczbową o odpowiedniej postaci całkowitej, pojedynczej lub podwójnej precyzji.

MKI$ (*wyrażenie_całkowite*)
MKS$ (*wyrażenie_pojedynczej_precyzji*)
MKD$ (*wyrażenie_podwójnej_precyzji*)
Dokonuje konwersji wartości liczbowej w wartość tekstową zapisywaną do pliku o dostępie bezpośrednim, zachowując odpowiednią postać całkowitą, pojedynczej lub podwójnej precyzji.

Instrukcje

OPEN *tryb*, # *np*, "*nazwa_pliku*" [,*dług_rekordu*]
Otwarcie pliku określonego *nazwą_pliku* w odpowiednim trybie zadanym jednoznakowym tekstem ("O", "I", "R") z przypisaniem plikowi otwartemu *numeru_pliku* logicznego z zakresu 1...15. Długość rekordu jest wyrażeniem całkowitym dla plików o dostępie bezpośrednim

CLOSE [# *np*[,# *np*] ...]
Zamknięcie współpracy z plikami o podanych numerach logicznych.

INPUT # *np*, *zmienna* [,*zmienna*] ...
Przypisuje zmiennym z listy zmiennych wartości odczytywane z pliku sekwencyjnego o podanym numerze pliku.

LINE INPUT # *np*, *zmienna_tekstowa*
Przypisuje *zmiennej_tekstowej* wartość tekstową odczytaną z pliku sekwencyjnego o podanym numerze pliku.

PRINT # *np*, [USING "*wzorzec*";] *lista_wyrażeń*
Wyprowadza wartości z *listy_wyrażeń*, według identycznych jak dla instrukcji PRINT zasad, do pliku sekwencyjnego o podanym numerze pliku.

WRITE # *np*, *wyrażenie* [, *wyrażenie*]...
Wyprowadza wartości z listy *wyrażeń*, według identycznych jak dla instrukcji WRITE zasad, do pliku sekwencyjnego o podanym numerze pliku.

FIELD # *np*, *szer_pola* AS *zm_tekst* [, *szer_pola* AS *zm_tekst*]...
Definiuje długość i strukturę bufora dla współpracy z plikiem o dostępie bezpośrednim.

LSET *zmienna_tekstowa = wyrażenie_tekstowe*
RSET *zmienna_tekstowa = wyrażenie_tekstowe*
Wprowadza do bufora współpracującego z plikiem o dostępie bezpośrednim wartości wyrównując je lewostronnie lub prawostronnie (RSET).
PUT # *np*, [*numer_rekordu*]
Zapisuje zawartość bufora do pliku o dostępie bezpośrednim umieszczając go w kolejnym lub wskazanym numerem rekordzie.
GET # *np*, [*numer_rekordu*]
Odczytuje z pliku o dostępie bezpośrednim kolejny lub wskazany numerem rekord zapisując go do bufora.

Ładowanie, składowanie i wiązanie programów

Instrukcje

SAVE "*nazwa_pliku*" [,A]
SAVE "*nazwa_pliku*" ,P, "*hasło*"
Składowanie tekstu programu zapisanego aktualnie w pamięci RAM do pliku o podanej nazwie. Postać składowana może być binarna, znakowa (opcja A) lub binarnie zakodowana z *hasłem* protekcji (opcja P)
LOAD "*nazwa_pliku*" [,R] [,P, "*hasło*"]
Ładowanie tekstu programu do pamięci RAM z pliku o podanej nazwie. Przy ładowaniu pliku składowanego uprzednio z protekcją wymagane jest podanie *hasła*. Opcja R powoduje natychmiastowe uruchomienie załadowanego programu.
RUN "*nazwa_pliku*" [,R] [,P, "*hasło*"]
Działanie podobne do działania polecenia LOAD z drobnymi różnicami dotyczącymi aktualnie otwartych plików.
COMMON *zmienna* [, *zmienna*] ...
Przechowywanie wartości podanych na liście zmiennych dla następnego programu załadowanego poleceniem CHAIN.
CHAIN [MERGE] "*nazwa_pliku*" [, *linia*] [,ALL] [,DELETE *zakres*]
Załadowanie kolejnego programu z podanego *nazwą_pliku* z zachowaniem części linii dotychczas istniejącego programu oraz części lub wszystkich wartości zmiennych.
MERGE "*nazwa_pliku*"
Doładowanie do istniejącej w pamięci RAM treści programu dodatkowych linii z instrukcjami ze wskazanego *nazwą_pliku.*
KILL "*nazwa_pliku*"
Usunięcie z pamięci pomocniczej pliku wskazanego *nazwą_pliku.*
NAME "*nowa_nazwa_pliku*" AS "*stara_nazwa_pliku*"
Zmiana nazwy pliku w pamięci pomocniczej.

Obsługa sytuacji wyjątkowych

Zmienne standardowe

ERR

Przechowuje kod błędu,

ERL

Przechowuje numer linii z wykrytym błędem.

Instrukcje

ERROR *wyrażenie_całkowite*

Generuje sytuację wyjątkową przypisując zmiennej ERR kod błędu określony wartością wyrażenia całkowitego.

ON ERROR GOTO *numer_linii*

Określa miejsce obsługi błędów wykrywanych w kolejno wykonywanych instrukcjach do chwili zmiany tego określenia lub zablokowania programowej obsługi błędów instrukcją ON ERROR GOTO 0.

RESUME 0
RESUME NEXT
RESUME *numer_linii*

Powrót do podstawowej sekwencji wykonywania instrukcji programu po zakończeniu obsługi sytuacji wyjątkowej – do linii z błędną instrukcją, następnej linii (NEXT) lub do wskazanej.

Łącze z zapisem wewnętrznym programu

Funkcje wspomagające

VARPTR (*nazwa_zmiennej*)
VARPTR (# *numer_pliku*)

Zwraca adres zmiennej lub początku bufora, związanego z danym otwartym plikiem.

USR [*cyfra*] (*wyrażenie*)

Zwraca wartość wyliczoną przez podprogram zapisany w języku wewnętrznym pod numerem określonym cyfrą z przedziału 0...9.

INP (*numer_portu*)

Udostępnia wartość bajtu odczytanego z portu o podanym numerze.

PEEK (*adres_bajtu*)

Udostępnia wartość bajtu pamięci o podanym adresie.

Instrukcje

DEF USR [*cyfra*] = *adres_podprogramu*

Definiuje *adres_podprogramu* przypisując go numerowi podprogramu wywoływanego jako funkcja USR.

CALL *nazwa_zmiennej* [(*argument* [,*argument*]...)]
Wywołanie podprogramu zapisanego w języku wewnętrznym pod adresem przypisanym zmiennej o podanej nazwie.

OUT *numer_portu, wyrażenie_całkowite*
Wyprowadzenie do portu o podanym numerze wartości pojedynczego bajtu.

WAIT *numer_portu, maska1* [,*maska2*]
Wstrzymanie działania programu do momentu spełnienia warunku określonego wartością wprowadzonego przez dany port bajtu i zadanych masek.

POKE *adres_bajtu, wyrażenie_całkowite*
Zapisanie pod danym *adresem_bajtu* pamięci RAM wartości określonej *wyrażeniem_całkowitym.*

D. Polecenia języka BASIC

CLOAD/48
CONT/72
CSAVE/48
DATA/63
DEF FN/190
DELETE/46
DIM/131
END/71
ERASE/148
FOR...TO...STEP/110
GOSUB/195
GOTO/107
IF THEN/96, 175
INPUT/64
LET/61, 85
LINE INPUT/161
LIST/46
LPRINT/67
MID$/171
NEW/47
NEXT/110
OPTION BASE/148
PRINT/67
READ/63
REM/71
RESTORE/64
RETURN/195
RUN/41
STOP/72
TRON, TROFF/226

E. Spis informacji podstawowych

O komputerze

Komputer sam nie myśli/11
Komputer osobisty/25, 28
Pamięć komputera/28
Użytkowanie komputera osobistego/43
Translator/29
System BASIC/40

O programowaniu

Algorytm/12
Program/17
Elementy składowe programu/19
Moduły programu/19
Struktura programu/20
Programowanie komputera/23
Metody konstruowania programu/24
Fazy życia programu/30
Składnia języka/29
Semantyka języka/29
Dostęp do kolejnych elementów wektora/134
Dostęp do kolejnych elementów macierzy/141
Tablica trójkątna/145
Podział logiczny zmiennych podprogramu/202
Testowanie programu/222
Usuwanie błędów z programu/224
Instrumentacja programu/225
Klasy błędów programu/217

Błąd składni/217, 218
Błąd wykonywania/217, 219
Błąd logiki/217, 222
Efektywność programu/106
Kopiowanie programów/49

O BASICu

Język BASIC/32
Zakres prezentacji języka BASIC/34
Polecenie/40
Zlecenie/40
Instrukcja/40
Składnia linii/41
Zbiór znaków języka/55
Zapis tekstów programów/56
Elementy języka BASIC/57
Typy wartości/57
Zakres wartości/57
Stała/58
Stałe typu liczbowego/59
Identyfikator zmiennej/60
Zmienne typu liczbowego/60
Zmienne typu tekstowego/60
Rozmieszczanie wartości w wierszu/67
Zapis wyrażenia/80
Kolejność obliczania wartości wyrażenia/80
Wywołanie funkcji standardowej/83
Wyrażenie całkowite/87
Wyrażenie relacji/94, 173
Wynik wykonania operacji relacji/95
Cykl/107
Wykonywanie cyklu/108
Wykonywanie cyklu wyliczanego/111
Zagnieżdżone instrukcje cyklu/115
Warunki zagnieżdżania instrukcji cyklu wyliczanego/116
Deklaracja tablicy/132
Nazwa zmiennej indeksowanej/132
Indeks/136
Położenie zmiennej indeksowanej w macierzy/143
Wartość typu tekstowego/159
Długość tekstu/159
Wyrażenia tekstowe/160

Wywołanie standardowej funkcji tekstowej/164
Porównywanie tekstów/172
Wywoływanie funkcji definiowanej/192
Kolejność wyliczania wartości funkcji definiowanej/193
Matematyczne funkcje standardowe/83
Tekstowe funkcje standardowe o wartościach liczbowych/164
Tekstowe funkcje standardowe o wartościach tekstowych/165
Znaki sterujące interakcją z systemem/42
Znaki sterujące i funkcyjne redagowania linii/44

F. Spis kanonów stylu programowania

Modularność programu/210
Struktura programu/74
Instrumentacja programu/229
Elegancki zapis programu/120
Komentarze/73
Zapis tekstu podprogramu/212
Dobór identyfikatorów/72
Zapis formuł w postaci wyrażeń/89
Zakaz wykorzystywania instrukcji GOTO/119
Wykorzystywanie instrukcji warunkowej/120
Instrukcje cyklu wyliczanego FOR...NEXT/120
Stosowanie tablic/151
Efektywne wykorzystywanie tablic/152
Wykorzystywanie funkcji tekstowych/186
Zestaw danych/73
Rzetelność danych/73

Skorowidz

ABS 83
algorytm 12, 13, 222
analiza tekstu 176
argument 79, 83, 164, 192
– funkcji 83, 192
– podprogramu 192
ASC 164
asembler 29
ATN 83
atrybut tablicy 130
– zmiennej 56

Błąd logiki 217, 222
– składni 52, 91, 217, 218
– wykonywania 51, 217, 219, 222

Centralna jednostka przetwarzająca 27
CHR$ 165
CLOAD 48
CONT 72, 226
COS 83
CSAVE 48
CTRL 159, 218
cyfry 55
– znaczące 57, 59, 67
cykl 107
– iteracyjny 107, 108
– wyliczany 107, 108
czas wykonywania operacji 90
część główna programu 196

Dane 19, 152
DATA 63
DEF FN 190
definicja funkcji 190, 221
– – rekurencyjna 193
definiowanie funkcji 190
deklaracja 19, 61, 148, 220
– dynamiczna tablicy 148
– obszaru na teksty 187
– tablicy 132, 154, 221
– standardowa 132, 148
DELETE 46
diagram 17
DIM 131
długość linii 41
– tekstu 159, 163, 171, 223
dokładność wyznaczenia wartości wyrażenia relacji 96
dostęp do elementów macierzy 141
– – – wektora 134
drukarka 26, 66
dzielenie przez zero 91

Edycja 218
efektywność obliczeń 89, 106, 139
– programu 106
ekran 26, 66, 166
elementy języka 57
– programu 19
– sąsiednie 134, 143
END 71, 213
ERASE 148
EXP 83

Fałsz 95, 172
fazy życia programu 30
FIX 83
FOR...TO...STEP 110
formuła matematyczna 82
funkcje arytmetyczne 82
– cyklometryczne 194
– logarytmiczne 85
– matematyczne 83
– standardowe 82
– tekstowe 164, 174
– trygonometryczne 85

Głośnik 26
GOSUB 195
GOTO 107, 225

Identyfikator 72
– funkcji 191
– zmiennej 60, 218
– – tablicowej 131
IF...THEN 96, 175
indeks 130, 136, 137, 154
INPUT 64, 74
INSTR 164
instrukcja 19, 40
– cyklu wyliczanego 110, 134, 142
– danych 63
– definicji funkcji 190
– deklaracji bazy 148
– – tablicy 131
– komentarza 71, 120
– końca cyklu wyliczania 71, 110, 219
– powrotu z podprogramu 195, 219
– przejścia 107
– – do podprogramu 195
– przewinięcia strumienia 64
– przypisania 61, 85
– skoku 107
– śledzenia 225, 226
– warunkowa 107, 171, 175, 198
– wczytywania 63
– wprowadzania 64, 74
– wstrzymywania 72
– wymazywania tablic 148
– wymiany tekstu 170
– wyprowadzania 66
– wznowienia 225
instrumentacja programu 225

INT 83
interpretator 29, 218, 220
istnienie w tekście wycinka 163
iteracje 223

Język asemblerowy 29
– BASIC 32
– programowania 17, 30

Kb 27, 159
klasy błędów 217
klawiatura 26
kod ASCII 159, 173, 231
– znaku 26, 159, 166
kodowanie 17, 100, 222
kolejność obliczania wartości funkcji definiowanej 193
– – – wyrażenia 80
kolumna 26
– tablicy 130
kompilator 30
komputer osobisty 11, 25, 28, 37
koniec linii 41
konkatenacja 160
kopiowanie programów 49
krok 110, 111
kursor 44

LEFT$ 165
LEN 164
LET 61, 85
liczby rzeczywiste, niewymierne, całkowite 58
– zmiennoprzecinkowe 57
LINE INPUT 161, 176
LIST 46, 218
lista nazw tablic 148
– wartości 63, 67, 218
– zmiennych 63, 64
litery 55
LOG 83
Logika programu 219
LPRINT 67

Ładowanie 48

Macierz 130
magnetofon 26, 48

mantysa 58
metoda rozwijania pętli 153
– ściskania pętli 153
metody konstruowania programu 24
MID$ 165, 171
mikrokomputer 27
mikroprocesor 27
moduły programu 19
monitor 26

Nadmiar 91, 106
napis 59, 159
– pusty 61
nazwa funkcji 164
– pliku 48
– zmiennej 59
– – indeksowanej 132
NEW 47, 226
NEXT 110
numer linii 40, 41, 46, 52, 64, 72, 75, 96, 107, 122, 195, 218, 225

Obcięcie wartości 84
obliczanie wyrażenia relacji 120
od ogółu do szczegółu 24, 196, 211
od szczegółu do ogółu 24
operator 56, 91, 218
– arytmetyczny 79
– relacji 95, 173
– tekstowy 160
OPTION BASE 148

Pamięć 28
– kasetowa 47
– operacyjna 25, 28
– pomocnicza 26
– stała (ROM) 28
– zmienna (RAM) 28
parametr 57, 191
– wejściowy 202
– wyjściowy 202
podprogram 194
podwyrażenie 89
podział logiczny zmiennych podprogramu 202
polecenie 40
położenie zmiennej indeksowanej w macierzy 143
poprawianie tekstu 218
poprawność programu 221
porównywanie tekstów 172
postać liczby 58
poziom edycji 218
– wprowadzania poleceń 40, 41, 52, 71, 75, 218
prawda 95, 172
PRINT 67
priorytet operatora 79, 95
procesor 25, 27
program 17
– błędny 10
– edycji 158
– zapętlony 71
programowanie komputera 23
przekształcanie tekstu 173
przerwanie zlecenia 42, 75
przetwarzanie tekstów 158
przypisanie wartości 193

READ 63
redagowanie tekstu linii 45
rekurencja 193
REM 71
RESTORE 64
RETURN 195, 199
RETURN (klawisz) 64, 161
RIGHT$ 165
rozmiar 130, 131
rozmieszczenie wartości w wierszu 67
RUN 41, 72, 225, 226

Schemat Hornera 89
SGN 83
semantyka języka 29
separator 56, 67, 218
sieć działań 14, 98, 109, 223, 228
SIN 83
składnia instrukcji 218
– języka 29
– linii 41, 43
składowanie 48
składowe programu 19
słowo kluczowe 41, 56, 85, 218
SPACE$ 165
spacje 168
SQR 83
stała 19, 57, 58, 59, 159, 218

standard języka 10
standardowe funkcje matematyczne 83
– – tekstowe 164
STOP 72, 225
stos 196
strefa 67, 69
STRING$ 165
STR$ 165
struktura liniowa 198
– programu 10, 19, 20, 198
– rozgałęziona 198
– sterowania 116, 212
– – decyzyjnego 99
strumień danych 160
– – wewnętrznych 63, 224
– wyjściowy 67
styl programowania 10, 50, 224
symbol 55, 56
– zachęty 38
system BASIC 30, 37, 40, 192, 218
sytuacje błędne 10, 51, 219

Tablica 129, 148, 223
– jednowymiarowa 130
– dwuwymiarowa 130
– trójkątna 145
– wielowymiarowa 153
tabulacja 44
TAN 83
tekst programu 158
– zachęty 64, 161
tekstowe funkcje standardowe o wartościach liczbowych 164
– – – – – tekstowych 165
telewizor 26
testowanie programu 185, 211, 222, 223, 225
translator 29, 51, 111
TROFF 226
TRON 226
tryb bezpośredni 40, 67, 218, 225, 226
typ funkcji 191
– liczbowy 57, 67, 72, 85, 131, 169, 191, 218
– tekstowy 57, 67, 72, 132, 158, 174, 191, 218, 220
– wartości 57
– zmiennej 59

Urządzenie zewnętrzne 25
usuwanie błędów z programu 224
użytkowanie komputera 43

VAL 164
VERIFY 49

Warstwa tablicy 130
wartość 61, 67
– całkowita 84
– funkcji 195
– końcowa 108, 111
– początkowa 108, 111
– pośrednia 225
– pusta 159
– relacji (dokładność) 95
– typu liczbowego 169
– – tekstowego 159
– zmiennej 59, 225
warunki zagnieżdżania instrukcji cyklu wyliczanego 116
wektor 130
weryfikacja poprawności 221
wiersz 26, 69, 130
– tablicy 130
wprowadzanie wartości 64, 74, 135, 161, 176
wskaźnik wprowadzania 63
wybranie fragmentu tekstu 166
wycinek tekstu 163
wykładnik dziesiętny 58
wykonywanie cyklu 108
– – wyliczanego 111
– programu 148
wymiana wycinka tekstu 163
– zawartości zmiennej 61, 62
wymiar 130
wynik 19
– pośredni 225
– wykonania operacji relacji 95
wyprowadzanie wartości 67, 160
wyrażenie 85
– arytmetyczne 79, 87
– całkowite 87, 131
– liczbowe 87, 190
– relacji 94, 172, 173
– tekstowe 158, 160, 190
wyszukiwanie tekstu 169
wywołanie funkcji 91
– – definiowanej 192
– – standardowej 83

wywołanie funkcji standardowej tekstowej 164
wznowienie programu 72, 226
– zlecenia 42, 47

Zagnieżdżone instrukcje cyklu 115
zajętość pamięci 212, 219, 221
zakodowanie 17, 100
zakres prezentacji języka 34
– wartości 57
zaokrąglenie wartości 84, 87
zapętlenie programu 71
zapis tekstów programów 56
– wyrażenia 80, 160
zawieszenie wykonywania zlecenia 42
zbiór znaków języka 55
zerowanie systemu 42
zestaw danych 225
zlecenie 40, 218
zmienna 19, 57, 59, 61
– globalna 192
– indeksowa 132
– lokalna 192, 220
– sterująca 108, 110, 111, 134, 225
– tablicowa 130
– tekstowa 161, 174
– typu liczbowego 60
– – tekstowego 60
znak funkcyjny 26, 44, 55, 159, 166
– sterujący 26, 42, 44, 55, 56, 64, 75
– – interakcją z systemem 42

WNT Warszawa 1987. Wydanie 1.
Nakład 100 000 + 300 egz.
Ark. wyd. 15,2. Ark. druk. 16,0.
Format B-5. Papier offset kl. III 70 g.
Pracę oddano do składu w marcu 1986 r.
Podpisano do druku we wrześniu 1987 r.
Druk ukończono w październiku 1987 r.
Symbol Et/82136/WNT
Drukarnia
im. Rewolucji Październikowej,
Warszawa. Zam. 2582/11/86. K-13

20867

Cena zł 450.–

ISBN 83-20-0834-2